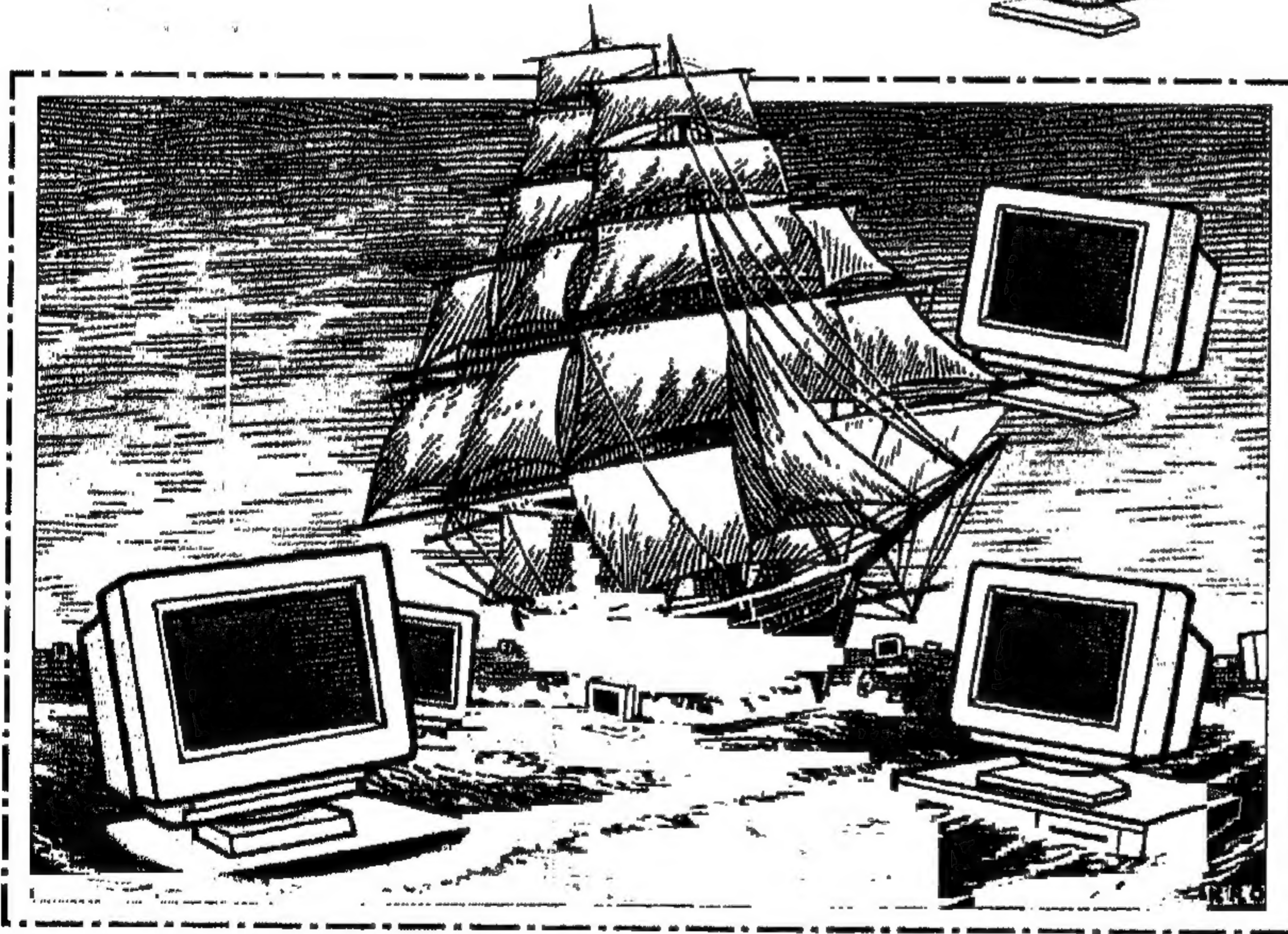


# محاضرات كليبر

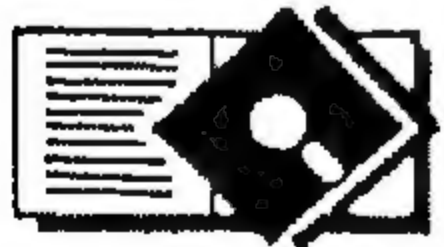
## Clipper Course Notes

الجزء الأول: مقدمة البرمجة

جديد!  
الإصدار 5.2



كتاب



قرص







# محاضرات كليبر

## Clipper Course Notes

الجزء الأول: مقدمة البرمجة

سليمان بن عبد الله الميمان

الأستاذ/ أحمد فراس مهاني

الدكتور/ محمد سعيد دباس

النشر والتوزيع:

الميمان للنشر والتوزيع

ص.ب: ٩٠٠٢٠ - الرياض ١١٦١٣

هاتف: ٤٠٢١٢١٩ - ٤٠٢٦١٩٤

فاكس: ٤٠١٤٩٩٦

محاضرات كليبر 5.2: مقدمة البرمجة

الطبعة الأولى - الرياض - ١٤١٥ هـ

### حقوق الطبع محفوظة

حقوق الطبع والنشر محفوظة لدار الميمان للنشر والتوزيع، ولا يحق لأي شخص نشر هذا الكتاب أو أي جزء منه أو تصويره أو إعادة طبعه أو تخزين محتوياته أو نقلها بأي وسيلة إلا بعد الحصول على إذن خطي وصريح مكتوب من الناشر.

### تنويه

تم إعداد هذه المحاضرات بالتعاون مع مؤسسة جرمبفيس الأمريكية المتخصصة في إعداد برامج تعليم لغة كليبر. وهذه المؤسسة معتمدة من قبل شركة Computer Associates، المالك الرسمي لجمع لغة كليبر 5.X.





المختصات





١٥	تمهيد
١٧	قواعد البيانات
١٧	إنشاء قواعد البيانات
١٨	اسم الحقل Field Name
١٨	نوع الحقل Field Type
١٩	أنواع الحقول المتوفرة
١٩	طول الحقل Field Length
٢٠	موقع الفاصلة العشرية في الحقل
٢١	تسمية قواعد البيانات
٢١	تصميم قاعدة البيانات
٢٣	الإسناد إلى حقول قاعدة البيانات
٢٣	حقول المذكرة Memo Fields
٢٦	فتح قواعد البيانات وإغلاقها
٢٧	الوظيفة dbUseArea() و dbCloseArea()
٢٨	ملفات الفهرسة Index Files
٢٩	مناطق العمل Work Areas
٣١	تحريك مؤشر السجل
٣٢	الأمر dbGoBottom() / GO BOTTOM
٣٢	الأمر dbGoTop() / GO TOP
٣٢	الأمر dbGoto(<record>) / GOTO <record>
٣٢	الأمر dbSkip([<count>]) / SKIP [<count>]
٣٣	الأمر dbSeek(<exp>) / SEEK <exp>
٣٥	الأمر _dbLocate() / LOCATE <exp>
٣٦	الأمر _dbContinue() / CONTINUE
٣٧	الأوامر والوظائف الأخرى لقاعدة البيانات
٣٧	الأمر dbAppend() / APPEND BLANK
٣٨	الأمر dbCommitAll() / COMMIT
٣٨	الأمر dbDelete() / DELETE

٣٩ .....	dbCreateIndex( ) / INDEX ON	الفهرسة
٤٣ .....	__dbPack( ) / PACK	الأمر
٤٣ .....	dbRecall( ) / RECALL	الأمر
٤٣ .....	dbReindex( ) / REINDEX	الأمر
٤٤ .....	dbSelectArea(<n>) / SELECT <n>	الأمر
٤٤ .....	dbSetFilter(<expr>) /SET FILTER TO <expr>	الأمر
٤٥ .....	dbClearFilter( ) / SET FILTER TO	الأمر
٤٥ .....	dbSetIndex(<n>) [ ADDITIVE] /SET INDEX TO <n>	الأمر
٤٥ .....	dbClearindex( ) / SET INDEX TO	الأمر
٤٦ .....	dbSetOrder(<n>) /SET ORDER TO <n>	الأمر
٤٦ .....	dbSetRelation( ) / ...SET RELATION TO...INTO	الأمر
٤٧ .....	dbClearRel( ) / SET RELATION TO	الأمر
٤٧ .....	dbUnlock( ) / UNLOCK	الأمر
٤٧ .....	dbUnlockAll( ) / UNLOCK ALL	الأمر
٤٧ .....	__dbZap( )/ZAP	الأمر

## ٤٩ ..... منهج ونظام الشبكة

٤٩ .....	SET EXCLUSIVE	الأمر
٤٩ .....	RLOCK( )	الوظيفة
٥٠ .....	FLOCK( )	الوظيفة
٥٠ .....	استراتيجيات قفل السجل	
٥١ .....	Unlock	إزالة القفل
٥١ .....	Lock	كتابة - تحرير -
٥١ .....	Unlock	إزالة القفل

## ٥٣ ..... أنواع البيانات DATA TYPES

٥٣ .....	Arrays	المنظومات
٥٤ .....	NIL	العدم
٥٤ .....	Code Blocks	كتل الشيفرة
٥٦ .....	Objects	الأهداف

٥٧ .....	Encapsulation الكبسلة
٥٧ .....	Inheritance الوراثة
٥٨ .....	طبقات الهدف في كليبر ٥,٢
٥٨ .....	اختبار أنواع البيانات Testing Data Type

## ٦١ .....

## العوامل OPERATORS

٦١ .....	Mathematical Operators العوامل الرياضية
٦٢ .....	Exponentiation الأسس
٦٢ .....	Modulus (remainder) الباقي
٦٣ .....	MOD( ) الوظيفة
٦٤ .....	العامل " + "
٦٥ .....	العامل " - "
٦٦ .....	عاملا التزايد والتناقص
٦٧ .....	أسبقية العوامل الرياضية
٦٩ .....	Relational Operators المعاملات العلائقية
٧٠ .....	Equivalence المساواة ( " = " و " = " )
٧١ .....	أولوية المعاملات العلائقية
٧١ .....	Logical Operators المعاملات المنطقية
٧٢ .....	Logical Short-Circuiting الاختصار المنطقي
٧٣ .....	أولوية المعاملات المنطقية
٧٣ .....	Assignment Operators معاملات التعيين
٧٤ .....	line Assignment-In التعيين المباشر
٧٥ .....	التعيين البسيط مقابل التعيين المباشر
٧٦ .....	Compound Assignment Operators معاملات التعيين المركب
٧٦ .....	أولوية معاملات التعيين
٧٧ .....	Special Purpose Operators معاملات الأغراض الخاصة
٧٧ .....	Macro Operator مُعامل ماكرو
٧٩ .....	تجميع كتل الشيفرة أثناء التشغيل
٨٠ .....	Alias Operator معامل النسخة المكافئة



٨٢ .....	تمرير المتغيرات بالإشارة <b>Pass by Reference</b>
٨٤ .....	القوسان الحاصران { } .....
٨٤ .....	القوسان المعقوفان [ ] .....
٨٥ .....	القوسان ( ) .....
٨٧ .....	<b>التجميع COMPILING</b> .....
٨٧.....	القاعدة اللغوية <b>Basic Syntax</b> .....
٨٧.....	عملية التجميع .....
٨٨.....	خيارات المجمع .....
٨٩.....	مفتاح (/b) تضمين معلومات برنامج كشف الأخطاء .....
٨٩.....	مفتاح (/n) منع إجراء بداية التشغيل .....
٩٠ .....	مفتاح (/w) إصدار رسائل تحذيرية .....
٩١ .....	<b>الربط LINKING</b> .....
٩١.....	الإحلال <b>Overlays</b> .....
٩٢.....	برامج ربط من شركات أخرى .....
٩٣ .....	برنامج الربط (Blinker) .....
٩٣ .....	برنامج الربط ( Warplink ) .....
٩٥ .....	<b>أساليب إضافة الملاحظات</b> .....
٩٧ .....	<b>تراكيب التحكم CONTROL STRUCTURES</b> .....
٩٧.....	تركيب <b>IF...[ ELSE ]..ENDIF</b> .....
٩٨.....	تركيب <b>DO WHILE..ENDDO</b> .....
٩٩.....	تركيب <b>DO CASE..ENDCASE</b> .....
١٠٠ .....	تركيب <b>FOR..NEXT</b> .....
١٠١ .....	تركيب <b>BEGIN SEQUENCE..END SEQUENCE</b> .....

## ١٠٣..... تحديد مجال المتغيرات

١٠٤..... إعلان المتغير الخاص Private

١٠٥..... مجال المتغيرات الخاصة PRIVATE

١٠٦..... إعلان المتغير العام PUBLIC

١٠٧..... مجال المتغير العام PUBLIC

١٠٩..... إعلان المتغير المحلي LOCAL

١٠٩..... مجال المتغير المحلي LOCAL

١١١..... إعلان المتغير الساكن STATIC

١١٢..... مجال المتغير الساكن STATIC

١١٤..... المتغيرات الساكنة على عرض الملف File-Wide Static Variables

١١٥..... تأسيس المتغيرات الساكنة STATIC

## ١١٧..... CODING CONVENTIONS اصطلاحات البرمجة

١١٧..... المساحة الفارغة White Space

١١٧..... الإزاحة Indentation

١١٨..... الأحرف الكبيرة Copitalization

١١٨..... أسماء المتغيرات Variable Names

١١٩..... مختصرات الأوامر Command Abbreviations

١١٩..... الإعلانات Declarations

١١٩..... الملاحظات Comments

## ١٢١..... USER INTERFACE TOOLS أدوات واجهة المستخدم

١٢١..... الشاشة Screen

١٢١..... أمر SAY @...

١٢٣..... أمر BOX @...

١٢٥..... الأمر ?

١٢٦..... أمر ??

١٢٦..... الأمر CLS



١٢٦.....	وظيفة الانزلاق ( ) SCROLL
١٢٧.....	أمر المسح @...CLEAR
١٢٧.....	التحكم بالألوان Color Control
١٢٨.....	لوحة المفاتيح Keyboard
١٢٩.....	وظيفة مفتاح الإدخال ( ) INKEY
١٣٠.....	وظيفة آخر مفتاح ( ) LASTKEY
١٣٠.....	وظيفة المفتاح التالي ( ) Nextkey
١٣١.....	وظيفة التنبيه ( ) ALERT
١٣٢.....	الأمر @...GET
١٣٣.....	أمر القراءة READ
١٣٤.....	تدقيق البيانات المدخلة Validating Data Entry
١٣٥.....	عبارة WHEN
١٣٦.....	شرط Clause
١٣٧.....	عمليات القراءة المتداخلة Nested READs
١٣٨.....	أمر ضبط المفاتيح SET KEY
١٣٩.....	قوائم الاختيارات Menus
١٣٩.....	أمر "التوجيه" @...PROMPT
١٤٠.....	أمر الاختيار من قائمة الاختيارات MENU TO
١٤٢.....	أمر ضبط الرسالة SET MESSAGE
١٤٢.....	أمر ضبط اللف SET WRAP
١٤٣.....	وظيفة الاختيار ( ) ACHOICE
١٤٩.....	كتابة الوظائف المعرفة من قبل المستخدم
١٤٩.....	الوظائف مقابل الإجراءات Functions Vs. Procedures
١٤٩.....	وحدات البرامج Modularity
١٤٩.....	التنظيم والتنسيق Housekeeping
١٥٠.....	وظيفة ضبط الألوان ( ) SETCOLOR
١٥٠.....	وظيفة اختيار الألوان ( ) COLORSELECT
١٥١.....	وظائف كل من ( ) ROW( ) / COL( ) / SETPOS( )
١٥١.....	وظيفة ( ) RESETSCREEN و ( ) SAVESCREEN

١٥٣.....	وظيفة التجهيز SET( )
١٥٤.....	وظيفة SETCURSOR( ) (تجهيز المؤشر)
١٥٥.....	وظيفة تجهيز مفتاح SETKEY( )
١٥٦.....	مثال:
١٥٧.....	وظيفة SETBLINK( )
١٥٧.....	استقلالية طور الفيديو
١٥٨.....	وظيفة SETMODE( ) (ضبط الوضعية)
١٥٨.....	وظيفة MAXCOL( ) / MAXROW( )
١٦١.....	الوظائف الساكنة Static Functions
١٦٥.....	شاشات إدخال البيانات
١٦٩.....	معالجة السلاسل والمذكرات
١٦٩.....	وظائف السلاسل
١٦٩.....	الوظائف ALLTRIM( ) و LTRIM( ) و RTRIM( )
١٧٠.....	الوظائف PADL( ) و PADC( ) و PADR( )
١٧١.....	وظيفة النسخ REPLICATE( )
١٧٢.....	الوظيفة SPACE( )
١٧٢.....	الوظيفة SUBSTR( )
١٧٣.....	وظيفة LEFT( ) و RIGHT( )
١٧٣.....	وظيفة UPPER( ) و LOWER( )
١٧٤.....	وظائف المذكرة MEMO( )
١٧٤.....	وظيفة تحرير المذكرة MEMOEDIT( )
١٧٧.....	وظيفة قراءة المذكرة MEMOREAD( )
١٧٧.....	الوظيفة MEMOWRITE( )
١٧٨.....	الوظيفة MLCOUNT( )
١٧٩.....	الوظيفة MEMOLINE( )
١٨١.....	المخرجات OUTPUT
١٨١.....	الطابعة

١٨١.....	الوظيفة SET DEVICE
١٨١.....	الوظيفة SET PRINTER( )
١٨٢.....	الأمر SET CONSOLE
١٨٣.....	الأمر ? و ??
١٨٣.....	إخراج الورق EJECT
١٨٣.....	وظيفة PCOL( ) و PROW( )
١٨٥.....	وظيفة SETPRC( )
١٨٦.....	ملف آسكي ASCII
١٨٦.....	الأمر COPY TO
١٨٨.....	أمر "جهاز بديلاً" SET ALTERNATE
١٨٨.....	ملف قاعدة البيانات DBF
١٩١.....	متفرقات
١٩١.....	عمليات الملف
١٩١.....	أمر "التجهيز الافتراضي" SET DEFAULT
١٩٢.....	الوظيفة File( )
١٩٢.....	الوظيفة COPY FILE
١٩٣.....	الوظيفة FEREASE( )
١٩٣.....	الوظيفة FRENAME( )
١٩٤.....	الوظيفة FERROR( )
١٩٥.....	إنشاء ملف قاعدة بيانات بشكل سريع On-The-Fly
١٩٥.....	تعديل هيكل ملف قاعدة البيانات بشكل سريع On-The-Fly
١٩٦.....	إسترجاع معلومات دليل
١٩٨.....	اختيار الملفات باستخدام DIRECTORY( ) و ACHOICE( )
١٩٨.....	نظام التاريخ والتوقيت
١٩٩.....	إسترجاع البيئة
٢٠٠.....	ملخص

# تمهيد

نهنتك على اختيارك كليب 5.2 لتطوير برامجك. لاشك أن مجمع كليب ليس له نظير عندما يكون الغرض هو كتابة برامج لإدارة قواعد البيانات وخاصة تلك المتوافقة مع dBASE III Plus.

إن المحاضرات المدونة في هذا الكتاب ستساعدك على التعلم والعمل السريع مع كليب 5.2 خلال وقت قصير. كما أن هناك العديد من المميزات الجديدة في الإصدار 5.2 تتعدى نطاق هذه المحاضرات ، وخاصة في مجال object-oriented ، إلا أنه لايلزمك معرفة كل جانب من جوانب كليب 5.2 لكي تستطيع كتابة برامج متخصصة وتعمل بقوة وفعالية ، على الأقل في هذه المرحلة المبكرة من التعلم إلى أن تتقن أساسيات كليب تماماً.





## قواعد البيانات

إن الهدف من كليبر هو نفس الهدف ذاته المقصود من dBASE III Plus ، ألا وهو إدارة المعلومات المخزونة في قواعد البيانات. في حين أن قواعد البيانات هي مجموعة من السجلات Records. وكل سجل Record هو مجموعة أو أكثر من الحقول Fields. إن أبسط فكرة لاستيعاب مفهوم قاعدة البيانات هي دليل الهاتف، أو مفكرة الهاتف التي يحملها معظم الناس في جيوبهم. فأولاً: يجب أن تعتبر مفكرة الهاتف قاعدة البيانات database. وبعد ذلك: تعتبر كل شخص في مفكرة الهاتف يمثل سجلاً record في قاعدة البيانات. وأخيراً تقسيم معلومات كل شخص إلى فئات كالاسم والعنوان والمدينة والبلد والرمز البريدي ورقم الهاتف. كل واحد من هذه الفئات السابقة يمكن اعتباره حقلاً Field.

### ملاحظة

إذا صادف وأن عملت فيما مضى على قاعدة بيانات أخرى غير dBASE III Plus ، فإن مصطلح سجل Record سيكون مرادفاً في هذه الحالة لجدول Table.

## إنشاء قواعد البيانات

يمكنك إنشاء قواعد البيانات بالعديد من الطرق ، بما فيها استخدام نقطة محث قاعدة البيانات dBASE ، أو برنامج الخدمات DBU الموجود مع رزمة برنامج كليبر 5.2 ، كما أن هناك حشداً كبيراً من البرامج التي يطلق عليها Shareware أو Public domaine. ونحن نقترح عليك بقوة استخدام برنامج الخدمات المجاني Shareware المسمى dBMAX والموجود على القرص المرافق لهذه المحاضرات.

وعند استخدام كليب 5.2 يجب أن تتأكد دوماً من أن أسماء قواعد البيانات تحتوي فقط على حروف وأرقام فقط ، على أن تبدأ هذه الأسماء دائماً بحرف واحد على الأقل. وعند مخالفة هذه القاعدة ستظهر رسالة مفادها: "Illegal characters in alias" (رموز غير مقبولة في الاسم المستعار) أثناء التنفيذ.

لأنهم الطريقة أو الأسلوب الذي تستخدمه لإنشاء قاعدة البيانات ، فإن أول شيء يجب عليك القيام به قبل أي شيء آخر هو تعريف البنية الهيكلية لقاعدة البيانات. وعندما يتم ذلك فإن عملك سيأخذ بشكل أساسي أسلوب الرجوع إلى الوراثة من الحقول Fileds إلى إنشاء قالب لكل سجل.

في حين أن كلاً من هذه الحقول يتكون من أربعة عناصر : Name ، و Data Type ، و Length ، و Decimals وهذا الأخير هو للاستخدام مع الأرقام فقط.

### اسم الحقل Field Name

يجب أن يكون طول اسم الحقل في قاعدة البيانات ما بين واحد إلى عشرة حروف. كما يمكن أن يحتوي على حروف وأرقام وشرطة سفلية " \_ " ، ولكن يجب أن يبدأ اسم الحقل بحرف واحد على الأقل. كما يجب أن تكون كل أسماء الحقول جميعها في قاعدة البيانات الواحدة متميزة عن بعضها وغير متشابهة.

### نوع الحقل Field Type

يوفر كليب 5.2 خمسة خيارات خاصة بأنواع حقول البيانات. وباستثناء حقل المذكرة Memo ، فإن كل نوع من هذه الأنواع له طول ثابت. وهذا يعني أن قاعدة البيانات تحتجز المقدار ذاته من المساحة لكل حقل في كل سجل. فعلى سبيل المثال ، لا يمكن حجز مساحة بطول ١٠٠ حرف لسجل واحد وبقيّة السجلات ٦٠ حرفاً.

## أنواع الحقول المتوفرة

- الحقل الحرفي Character - وهي البيانات التي تمثل بشكل خليط من الحروف والأرقام (كالأسماء والعناوين وأرقام الهواتف وغير ذلك). وستجد أن هذا النوع من البيانات هو أكثر الأنواع شيوعاً وملاءمة لمعظم حقول قواعد البيانات.
- الحقل الرقمي Numeric - وهي البيانات التي يجب أن تستخدم في إجراء العمليات الحسابية. (كالتوسط الحسابي والجمع وغيرهما).
- حقل التاريخ Date - وهي البيانات التي تمثل التاريخ (كتاريخ الفاتورة ، أو تاريخ الميلاد).
- الحقل المنطقي Logical - وتمثل بيانات هذا النوع على صورة صائب "حقيقي" True ، أو "غير حقيقي" False. فعلى سبيل المثال ، إن الحقول المنطقية هي أعلام يمكن استخدامها للإشارة إلى أن سجل العميل نشط أو لا ، أو هل الصنف لا زال موجوداً في المستودع أم لا ، وغير ذلك. وسنناقش لاحقاً حقول المذكرة Memo بتوسع وبتفصيل أكثر.

وعندما تنتقل إلى البنية الهيكلية لقاعدة البيانات ، نجد أن كل نوع حقل يشار إليه بصورة عامة بالحروف الأولى من اسم النوع (C / N / D / L / M).

## طول الحقل Field Length

عندما تقوم بتعيين حقل الحرف والتاريخ يجب أن تحدد طول الحقل. وأما الحقول الثلاث الأخرى فهي مزودة بطول افتراضي لكل منها ( فالتاريخ طوله ٨ والمنطقي طوله ١ والمذكرة ١٠ ). كما ينبغي أن تلاحظ أن طول المذكرة Memo غير متصل بهذا الموضوع ، حيث يمكن أن يصل طول الحقل كأقصى حد إلى ٦٤ كيلو بايت.

## موقع الفاصلة العشرية في الحقل

يمكنك تحديد عدد الأرقام بعد الفاصلة العشرية المطلوب حجزها للحقول الرقمية. لاحظ أنك إذا اخترت القيام بهذا العمل يجب أن تحجز موقعاً إضافياً خاصاً بالفاصلة العشرية. فعلى سبيل المثال ، إذا خططت أن يكون لديك رقم حده الأقصى له هو 99999.99 في قاعدة البيانات ، فإن الطول اللازم تعيينه لمثل هذا الرقم هو ٨ والأرقام العشرية ٢.

لقد تم ضبط الأرقام بعد الفاصلة العشرية بالنسبة لبقية أنواع البيانات بالرقم 0.  
مثال:

إن البنية الهيكلية لقاعدة البيانات الخاصة في دليل الهاتف يجب أن تأخذ الشكل التالي:

Fieldname	Type	Length	Decimals
NAME	C	25	0
ADDRESS	C	50	0
CITY	C	30	0
STATE	C	2	0
ZIP	C	5	0
PHONE	C	10	0

ستلاحظ أن حقل الهاتف Phone من النوع الحرفي Character Type. ونحن ننصح بذلك حتى وإن كان الحقل لا يحتوي إلا على أرقام فقط. فإذا لم يكن تنو استخدام هذا الحقل في إجراء العمليات الحسابية فينبغي عليك تعريفه ليكون من النوع الحرفي ، فإن التعامل مع الحقول الحرفية أسهل في إنشاء ملفات الفهرسة من الرقمية (وسنناقش هذا الموضوع فيما بعد). لاحظ أيضاً أنه بعد كل هذا ، فإن طول حقل الهاتف عشر بايتات، وهي مساحة غير كافية لأن يحتوي الحقل على شروط أو أقواس وخلافه. في حين أننا لسنا بحاجة لحجز أية مساحة لوضع الشروط أو الأقواس أو أي نص ثابت آخر ، وذلك لأنه يمكن إضافة مثل هذه العلامات في فقرة الصورة Picture Clause. سنناقش فقرة الصورة في وقت لاحق من الكتاب.

## تسمية قواعد البيانات

ومع أنه يمكن إعطاء اسم ملف قاعدة البيانات أية ثمانية حروف ، إلا أننا ننصح بأن تتبع القواعد ذاتها المستخدمة في تسمية الحقول (وهي خليط من الحروف والأرقام على أن يبدأ الاسم بحرف واحد على الأقل).

وبشكل تلقائي فإن ملف قاعدة البيانات في كليبر سيأخذ الامتداد الخاص بقاعدة البيانات dBASE III Plus ذاته (DBF). كما أن لك الحرية المطلقة في استخدام أي امتداد ترغب فيه. وفي الواقع فإن العديد من مطوري برامج كليبر يفضلون استخدام امتدادات بديلة أخرى في محاولة منهم لإيقاف عبث مستخدمي برامج dBASE III Plus (حيث أن هذا قد يكون خطراً على البيانات في كثير من الأحيان وخاصة عندما تكون البيانات سرية) من الوصول إلى قاعدة البيانات من خارج تطبيقات كليبر.

## تصميم قاعدة البيانات

هناك العديد من المراجع المتخصصة في تصميم قواعد البيانات ، وبما أن لدينا الكثير من الموضوعات التي نريد الحديث عنها ، فإننا نكتفي بتقديم أمثلة موجزة مفيدة ، يا ذن الله تعالى ، للأشخاص الذين ليس لهم أية خبرة في تصميم قواعد البيانات.

لنفترض أن لديك قاعدة بيانات الفواتير وتريد أن يكون لديك سجل منفصل له سطر بند في الفاتورة. وكل فاتورة ستحمل تاريخ ورقم العميل المرتبط بها. وكل سطر بند يتكون من رقم القطعة ، والكمية ، بالإضافة إلى الحقل المنطقي الذي يشير إلى حالة البند ما إذا كان مفتوحاً أم لا.



ينبغي أن تضع هذه المعلومات جميعها داخل قاعدة البيانات ، كالتالي أدناه:

## Structure of INVOICES.DBF

Fieldname	Type	Length	Decimals
INVOICE_NO	C	10	0
CUST_NO	C	10	0
DATE	D	8	0
LINEITEM	C	10	0
QUANTITY	N	6	0
OPEN	L	1	0
APP. Record Size		45	

ومن ناحية أخرى ، فإن هذا يعني أنك بحاجة على الأقل لتكرار رقم العميل ، والتاريخ لكل سجل في سطر البند LINEITEM. إلا أنه يمكنك التخلص من هذه الزيادة بتقسيم البنية الهيكلية إلى قاعدتي بيانات ، واحدة خاصة بمعلومات الفاتورة والأخرى للمعلومات المتعلقة بالبند السطري.

## Structure of INVOICES.DBF

Fieldname	Type	Length	Decimals
INVOICE_NO	C	10	0
CUST_NO	C	10	0
DATE	D	8	0
APP. Record Size		28	

## Structure of LINEITEM.DBF

Fieldname	Type	Length	Decimals
INVOICE_NO	C	10	0
LINEITEM	C	10	0
QUANTITY	N	6	0
OPEN	L	1	0
APP. Record Size		27	

قد يبدو لك من النظرة الأولى أن قاعدة البيانات هنا أكبر من المذكورة في المثال السابق، وذلك لظهور حقل رقم الفاتورة INVOICE\_NO في كلا القاعدتين. إن ظهور هذا الحقل في كلا القاعدتين ضروري للغاية لإنشاء رابط بين هاتين القاعدتين. وعلى

هذا الأساس سيتم ربط السجلات. سيصبح استخدام قاعدة البيانات في غاية الروعة والفعالية وليس كاستخدام قاعدة بيانات مفردة تضم الحقول جميعها.

250 invoices, 10 lineitems per invoice

using INVOICE.DBF	2500 records x 45 bytes/record =	112,500 bytes
using INVOICE.DBF	250 records x 28 bytes/record =	7,000 bytes
and LINEITEM.DBF	2500 records x 27 bytes/record =	67,500 bytes
	Total =	74,500 bytes

## الإسناد إلى حقول قاعدة البيانات

لدى الإحالة إلى حقول قاعدة البيانات في تطبيقات كبير، ننصحك أن تستخدم خاصية الاسم المستعار alias لقاعدة البيانات. كما يمكنك عدم استخدامه بافتراض أن حقول قاعدة البيانات موجودة في منطقة العمل الحالية، ولكن هذه الطريقة ضعيفة وسيئة من الناحية التطبيقية إذ أن تطبيقات كبير المتواضعة والبسيطة يوجد فيها أكثر من منطقة عمل تكون مفتوحة في الوقت ذاته، فإذا لم تستخدم الأسماء المستعارة لكل حقل فستجد نفسك أمام مشكلة عويصة تحتاج معها إلى فريق للصيانة.

ويوضح المثال أدناه طريقة الإسناد إلى حقول قاعدة البيانات:

```
? customer->lastname  
customer->firstname := "Emad"
```

## حقول المذكرة Memo Fields

يشبه حقل المذكرة Memo Field البيانات الحرفية، حيث يمكن أن يحتوي على أية بيانات خليط من الحروف والأرقام بالإضافة إلى أنه أكثر حرية. فيمكن أن يتلاءم مع الكثير من الحالات التي يكون فيها استخدام الحقول الحرفية character fields غير مناسب.

ومن الأمثلة العملية على هذه الحالات متابعة الاتصالات التي يقوم بها العميل في كل مرة. فمن المستبعد طبعاً أن يكون لديك العدد ذاته من الاتصالات لكل عميل ، بل ربما لايجري بعض العملاء أي اتصال على الإطلاق. وبناء على ذلك ، فإنه لا توجد طريقة لمعرفة مقدار المساحة اللازمة لمثل هذا النوع من الحقول الحرفية. ولكن باستخدام حقول المذكرة سيكون بإمكانك استخدام المساحة الضرورية لكل سجل يستدعي مثل هذا النوع من المعلومات. وفي الوقت ذاته فأنت لا تجبر كليبر أن يقوم بحجز مساحة كبيرة أو غير معروفة القدر لكل سجل في قاعدة البيانات. وبهذه الطريقة سيتحدد مقدار المساحة بالمعلومات المطلوب تخزينها فيه.

وإذا قمت بتصميم حقل مذكرة Memo فإن كليبر سيقوم بإنشاء ملف خاص منفصل يحمل اسم ملف قاعدة البيانات DBF. ولكن الامتداد في هذه المرة سيكون DBT. ( وعلى خلاف قاعدة البيانات يجب استخدام الامتداد DBT. لملف الذاكرة ولا يمكن تغييره بأي حال من الأحوال). كما يجب أن يكون هذا الملف (أو الملفات) في دليل ملفات قاعدة البيانات ، وذلك لأن كليبر سيقوم بفتح ملف DBT. وملفات DBF. في الوقت ذاته.

عندما تريد اتخاذ قرار ما إذا كان الحقل المطلوب ينبغي أن يكون حقل سلسلة حرفية طويل أو حقل مذكرة ، فإنه يجب أن تفكر وتساءل نفسك (أو العميل) كم هو عدد السجلات المتوقع أن تحتاج إلى تلك المعلومات. فإذا تبين لك أن السجلات جميعها تقريباً (ولنفترض ٨٠٪ منها) عندما تستخدم الحقل الحرفي character field ، ولكن إذا كنت تظن أن عدداً قليلاً من السجلات سيحتاج إلى تلك المعلومات ، فإنه ينبغي استخدام حقل المذكرة MemoField. تأمل المقارنة التالية أدناه:

الخيار #١ : الحقل الحرفي ، الطول ١٠٠ .

حجم ١٠,٠٠٠ سجل في ملف DBF. = ١,٠١٠,٠٦٧ بايت.

الخيار #٢ : حقل المذكرة.

حجم ١٠,٠٠٠ سجل في ملف DBF = ١١٠,٠٦٧ بايت.

افترض أن حجم ملف DBT ١٠٪ = ٥١٢,٦١٣ بايت.

للسجلات المطلوبة لـ: ١٠٠ حرف من المعلومات.

الإجمالي = ٦٢٢,٦٨٠ بايت.

لاحظ أن ملف DBT أكبر من المتوقع فهو يعتمد على ١٠٠ حرف لكل ١٠٠٠ سجل Record ، وذلك لأن الكتلة التي حجزها حقل المذكرة تساوي ٥١٢ بايت. فعلى سبيل المثال ، إذا كان حقل المذكرة لسجل معين يبلغ ٥١٢ بايت فإن كليبر سيحجز له ١٠٢٤ بايت في ملف DBT.. ولكن مع هذا فإننا نفضل استخدام المذكرة بدلا من السلسلة الحرفية الطويلة ، لأننا بذلك نحفظ مايقارب ٤٠٪ من مساحة التخزين اللازمة.

ومع من أن حقل المذكرة مفيد – بكل تأكيد – إلا أن له جوانب أخرى سيئة والتي ينبغي أن نعرفها وهي:

- لايمكن استخدام حقل المذكرة كمفتاح key في الفهرسة ، وذلك لأن مفاتيح الفهرسة في كليبر لا بد أن تكون ذات طول ثابت ، وهذا غير متوفر في حقل المذكرة ، إلا أن هذا الأمر لايشكل كارثة.
- حقل المذكرة يتطلب من كليبر أن يفتح ملف إضافي للتعامل مع قاعدة البيانات يحتوي على حقل المذكرة. إلا أن هذا الأمر أيضا لايعد مشكلة في الوقت الحالي حيث أن نظام التشغيل DOS 3.3 فمافوق يسمح بفتح العديد من الملفات في الوقت ذاته دون أية مشكلة.
- المشكلة الأخيرة ، هي عندما تقوم بتحرير المذكرة لتعديل بعض الأمور فيها فإن المذكرة الأصلية ستظل باقية في ملف DBT. دائما مع المحتويات الجديدة. وهذا يعني

أن ملف DBT. سيصبح أكبر فأكبر تدريجياً. ويطلق على هذه الظاهرة بالذاكرة المنتفخة Memo bloat ، ويمكن تخاشي هذه الظاهرة بالطريقة التالية: القيام بشكل دوري ومنتظم بنسخ بنية قاعدة البيانات لإنشاء قاعدة بيانات فارغة ، ونسخ محتويات قاعدة البيانات الحالية للبنية الجديدة ، ثم إعادة تسميتها وفقاً لذلك. قطعة برنامج العرض التالي توضح هذه الطريقة:

```
USE datafile
COPY STRUCTURE TO scratch
USE scratch
APPEND FROM datafile
CLOSE DATA
RENAME datafile.dbf TO datafile.bak
RENAME datafile.dbf TO datafile.tbk
RENAME scratc.dbf TO datafile.dbf
RENAME scratc.dbf TO datafile.dbt
```

## فتح قواعد البيانات وإغلاقها

قواعد البيانات تصبح عديمة الفائدة إذا لم نستطع فتحها ، أو إغلاقها بشكل جيد. يمكننا القيام بكلا هذين العاملين باستخدام الأمر USE. فعلى سبيل المثال، يقوم سطر الأمر التالي بفتح قاعدة البيانات الخاصة بالعملاء:

```
use customer
```

ولإغلاق قاعدة البيانات الموجودة في منطقة العمل الحالية ، يمكنك ببساطة طباعة الأمر USE فقط لإنجاز هذه المهمة.

يتضمن الأمر USE على العديد من الشروط الاختيارية والتي يمكن استخدامها معه :

- SHARED – (مشارك) ويشير هذا الشرط إلى ضرورة فتح ملف قاعدة البيانات ليكون قابلاً للاستعمال المشترك على الشبكة.



- EXCLUSIVE – (خاص) وهذا الشرط عكس SHARED. لاحظ أن كلاً من الفقرة SHARED و EXCLUSIVE يستخدمان بشكل متبادل.
- READONLY – (قراءة فقط) يشير هذا الشرط إلى أنه يمكن عرض ملف قاعدة البيانات ولكن لا يمكن إجراء أية تعديلات عليه بأية طريقة.
- NEW – (جديد) افتح ملف قاعدة بيانات موجودة في منطقة العمل الأخرى.
- ALIAS – (اسم مستعار) يسمح هذا الشرط بتعيين اسم بديل (أو مستعار) يمكن استخدامه للدلالة على قاعدة البيانات الموجودة في برنامجك.
- INDEX – (فهرس) يسمح هذا الشرط بتعيين فهرس أو أكثر لفتح مع قاعدة البيانات.

وهنا تشاهد عدداً من الأمثلة على استخدام الأمر USE :

```
// افتح ملف الفواتير للعمل المشترك في منطقة العمل التالية //
use invoice new shared
// افتح ملف المورد للقراءة فقط في منطقة العمل التالية //
use vendor new readonly
// افتح ملف العملاء ثم عيّن الاسم "cust" كبديل لاسم ملف قاعدة البيانات //
use customer new alias cust
// أغلق قاعدة البيانات في منطقة العمل الحالية //
use
```

## الوظيفة dbCloseArea( ) و dbUseArea( )

إن هاتين الوظيفتين تكافئان الأمر USE الذي يستخدم لفتح قاعدة البيانات وإغلاقها على التوالي.

وقبل أن نذهب بعيدا ، ينبغي أن تعلم أن كليبر 5.2 لديه العديد من الوظائف المتكافئة. وفي معظم الأحوال لا توجد أية فروقات أو اختلافات بين استخدام الأمر أو الوظيفة المكافئة. وفي الأحوال جميعها يعود هذا الأمر إلى التفضيل الشخصي. ونعتقد أن من الضروري أن تطلع على الوظائف بالإضافة إلى الأوامر العادية ، وذلك لأن معرفة هذه الوظائف تسمح لك باستخدام عامل الاسم البديل alias operator لتجعل برنامجك ذاتي التوثيق (سنناقش عامل الاسم البديل (المستعار) بشكل مفصل لاحقاً).

## ملفات الفهرسة Index Files

إن أحد أعظم الفوائد من تخزين المعلومات في قواعد البيانات هي السرعة في عمليات البحث واسترجاع المعلومات. إن البحث السريع لا يتم بصورة تلقائية. فلنفترض أن لديك قاعدة بيانات تخص عملاء الشركة وترغب أن تقوم بفرز هذه القاعدة بواسطة اسم العائلة لكل عميل. فمن الأشياء البعيدة الاحتمال أن يكون قد تم إدخال البيانات مرتبة باسم العائلة لكل عميل ، لأن البيانات في العادة يتم إدخالها في أوقات متباعدة. فعلى سبيل المثال قد يكون لدينا السجلات التالية:

Record 1: Sameer  
Record 2: Ahmad  
Record 3: Reem  
Record 4: Kareem  
Record 5: Obeid

ومع أنه يمكن إجراء عملية فرز حقيقية لقاعدة البيانات اعتمادا على المعايير المعطاة ، إلا أنها ستكون بطيئة. ولذلك سنستخدم ملف الفهرسة index file لضمه إلى قاعدة البيانات لكي نستطيع مشاهدة البيانات بالترتيب المطلوب.

إن المعلومات التي سيتم تخزينها في ملف الفهرسة هي التي تم طلبها في معايير الفرز. فعلى سبيل المثال ، إذا قمت بفهرسة ملف العملاء بواسطة اسم العائلة ، والمعلومات التي سيتم تخزينها في ملف الفهرسة هي اسم العائلة الموجود في كل سجل

وبرفقته رقم السجل المطابق. ولهذا السبب فإن ملفات الفهرسة غالبا ما تكون أصغر من قاعدة البيانات المناظرة لها.

إن معايير الفرز الخاصة بملف الفهرسة يطلق عليها مفتاح الفهرسة index key. كما يمكن أن يتألف مفتاح الفهرسة من حقل بيانات وحيد أو مجموعة من الحقول. كما يجب أن تلاحظ أنه إذا قررت جمع عدد من الحقول في قالب مفتاح فهرسة واحد ، فإنه يجب عليك القيام بتحويل الجميع إلى نوع البيانات ذاته (وفي الغالب ما يكون هذا التحويل إلى النوع الحرفي Character).

إن الامتداد الذي تأخذه ملفات الفهرسة في كليبر هو NTX. وكما هو الحال بالنسبة لملفات قواعد البيانات لكل مطلق الحرية في تعيين الامتداد الذي ترغب فيه. كما يمكنك فتح ١٥ ملف فهرسة في الوقت ذاته كأقصى حد لكل قاعدة بيانات.

## مناطق العمل Work Areas

في أي وقت تقوم فيه بفتح قاعدة البيانات ، يعطي كليبر قاعدة البيانات هذه مناطق العمل الخاصة بها. في حين أن كليبر يسمح لك باستخدام ٢٥٠ ملف كحد أقصى تعمل في وقت واحد في مناطق عمل مختلفة ، وبناء على ذلك يمكنك فتح ٢٥٠ ملف قاعدة بيانات مختلف.

وكل منطقة عمل لها مجموعة معلومات خاصة بها تشتمل على : مؤشر السجل Record Pointer ، ومحدد بداية الملف Beginning-of-file marker ومحدد نهاية الملف end-of-file marker ، وشرط التصفية filter condition وشرط الموقع locate condition ، وعلم " السجل الموجود ". كما أن الوظائف التالية أدناه تسمح باستطلاع حالة منطقة العمل النشطة:

الوظيفة ( ) RECNO : تقوم هذه الوظيفة بإرجاع الموقع الحالي لمؤشر السجل في منطقة العمل.

الوظيفة ( ) BOF : إذا كنت في السجل الأول في قاعدة البيانات وتحاول تحريك مؤشر السجل إلى الوراء ، فإن محدد بداية الملف سيكون "حقيقياً" True. وعلى هذا فإن الوظيفة ( ) BOF تسمح لك بفحص حالة هذا العلم.

الوظيفة ( ) EOF : إذا كنت في السجل الأخير في قاعدة البيانات وتحاول تحريك مؤشر السجل إلى الأمام ، فإن محدد نهاية الملف سيضبط الوضع على True. وهذا لا يشبه بحال من الأحوال البداية على السجل الأخير لقاعدة البيانات. وإن نهاية الملف في الواقع وراء السجل الأخير. كما أن عملية البحث غير الناجحة ستجعل محدد نهاية الملف يضبط على True. إن الوظيفة ( ) EOF تسمح لك بأن تفحص حالة هذا العلم.

الوظيفة ( ) FOUND : عندما تحاول في أي وقت البحث في منطقة العمل ، فإن علم السجل الموجود record found flag سيضبط على الوضع "غير حقيقي" False. كما أن الوظيفة ( ) FOUND تسمح لك بفحص حالة هذا العلم.

الوظيفة ( ) LASTREC : تقوم هذه الوظيفة بإرجاع عدد السجلات الموجودة في منطقة العمل.

الوظيفة ( ) DBFILTER : إذا كان المرشح نشطاً في هذه المنطقة ، فإن هذه الوظيفة ستقوم بإرجاع سلسلة حرفية تحتوي على شرط ترشيح. (انظر أمر SET FILTER أدناه لمزيد من المعلومات).

يمكنك الاعتماد على هذه الوظيفة في تحديد سواء أتمت عملية البحث الأخير بنجاح أم لا، وما هو السجل الحالي ، الخ. تذكر أن كل منطقة عمل لها مجموعة المعلومات الخاصة بها ، فعلى سبيل المثال ، إذا كان هناك عشر مناطق عمل نشطة ، فمعنى هذا أن هناك

عشرة مؤشرات سجل ويمكن للوظيفة ( ) RECNO أن تعطيك كأقصى حد عشر قيم مختلفة.

إن أية أوامر أو وظائف مرتبطة بقواعد البيانات تأخذ مواقعها في منطقة العمل النشطة حالياً اللهم إلا إذا سبقت بالاسم البديل (المستعار) لقاعدة البيانات. يمكنك إنشاء مناطق عمل مختلفة باستخدام الأمر SELECT يعقبه البديل المطلوب. من ناحية أخرى ، كما ذكرنا أعلاه فإننا نحبذ أن تستخدم المعامل البديل للإشارة لمناطق العمل. إن استخدام الأمر SELECT يجعلك عرضة للأخطاء الدقيقة التي تحدث في منطقة العمل.

## تحريك مؤشر السجل

كما ذكرنا أعلاه ، فإن كل منطقة عمل لها مؤشر سجل خاص بها ، والذي يقوم بتدوين ومتابعة الحركة التي تقوم بها في كل قاعدة بيانات. إن الأوامر التالية أدناه تسمح بتحريك مؤشر السجل في منطقة العمل المعطاة.

### تنبيه

يقوم كليب 5.2 بإضافة فاحص للخطأ لكل أوامر تحريك مؤشر السجل أدناه أو وظائفه. لا تستخدم هذه الأوامر أو الوظائف دون أن يكون لديك قاعدة بيانات مفتوحة في منطقة العمل الحالية!



### الأمر dbGoBottom( ) / GO BOTTOM

يقوم هذا الأمر بتحريك مؤشر السجل للسجل الأخير في ملف قاعدة البيانات. فإذا كان ملف الفهرس index نشطاً ، فإن السجل الأخير سيعتمد على مفتاح الفهرس index key. وبخلاف ذلك ، فإن المؤشر سيعتمد السجل الأخير في قاعدة البيانات.

### الأمر dbGoTop( ) / GO TOP

يقوم هذا الأمر بتحريك مؤشر السجل للسجل الأول في ملف قاعدة البيانات. إذا كان الفهرس نشطاً ، فإن هذا الأمر سيعتمد على مفتاح الفهرس. وبخلاف ذلك سيعتمد المؤشر السجل الأول في قاعدة البيانات.

### الأمر dbGoto(<record>) / GOTO <record>

يقوم هذا الأمر بتحريك مؤشر السجل رقم سجل محدد. إن كون الفهرس نشطاً أو عدمه لا يؤثر في شيء.

إذا حددت رقم السجل <record> خارج النطاق ، فإن كلاً من العلم ( ) BOF و EOF( ) سيضبطان على True ، و يتحرك مؤشر السجل إلى نهاية الملف.

للشبكات: يمكنك استخدام ( ) GO RECNO لتجديد البيانات في السجل الحالي من القرص.

### الأمر dbSkip([<count>]) / SKIP [<count>]

يقوم هذا الأمر بتحريك مؤشر السجل إلى موقعه الحالي. إذا لم تحدد <count> فإنه يتم افتراضه سجلاً واحداً إلى الأمام. إذا كنت تستخدم مفتاح الفهرسة index key. فإن

مؤشر السجل سينتقل إلى السجل المنطقي التالي اعتماداً على مفتاح الفهرسة. أما إذا لم تستخدم الفهرس ، فعندئذ سينتقل مؤشر السجل ببساطة إلى السجل الطبيعي التالي.

إذا حاولت استخدام SKIP للقفز إلى الأمام متجاوزاً آخر سجل ، فإن مؤشر السجل سيقف في نهاية الملف وعلم الوظيفة ( ) EOF سيضبط على "حقيقي" True. أما إذا حاولت القفز إلى الوراء متجاوزاً أول سجل ، فإن مؤشر السجل سيقف في بداية الملف وسيتم ضبط علم الوظيفة ( ) BOF على True.

أما المتغير الاختياري <count> فهو يمكن أن يكون عدداً موجباً أو سالباً. فالأعداد الموجبة تسبب في التحرك إلى الأمام ، و السالبة ستتسبب في التحرك إلى الوراء.

### فكرة عن الشبكة

إن الأمر SKIP (بالإضافة إلى أية تحركات مؤشر سجل أخرى) سيجعل بالضرورة أيّ تغيرات في منطقة العمل الحالية مرئية لتطبيقات كليبز الأخرى التي يفترض أنها تشترك في قاعدة البيانات ، والتي كان فيها السجل الحالي مقفلاً بواسطة الوظيفة ( ) RLOCK. كما يمكنك أيضاً إجراء هذا التحديث بالقوة دون تحريك مؤشر السجل بإصدار الأمر .SKIP 0

### الأمر dbSeek(<exp>) / SEEK <exp>

يساعدك هذا الأمر على البحث بشكل سريع وفوري عن قطعة البيانات المحددة اعتماداً على الفهرس النشط active index. ويجب أن يكون لديك فهرس نشط لاستخدام هذا الأمر. فإذا تم إيجاد البيانات المطلوبة ، فسيتم ضبط علم الوظيفة ( ) FOUND على "حقيقي" True.

إذا فشلت عملية البحث ، فإن مؤشر السجل سيقف في أحد موضعين اعتمادا على نوع البحث. فهناك نوعان للبحث هما: "العادي" regular ، و "البحث الحساس softseek". فإذا فشلت عملية البحث العادية ، فإن مؤشر السجل سيقف في نهاية الملف (وراء السجل الأخير) وسيتم ضبط علم الوظيفة ( EOF ) على True.

إذا فشلت عملية البحث الحساسة softseek سينتقل مؤشر السجل إلى السجل المنطقي التالي في قاعدة البيانات ، وعندها لن يتم تغيير علم الوظيفة ( EOF ). دعنا نلق النظر مرة أخرى على قاعدة بيانات العملاء السابقة:

Record 1: Sameer  
Record 2: Ahmad  
Record 3: Reem  
Record 4: Kareem  
Record 5: Obeid

فإذا قمت بتنفيذ البحث العادي عن "Jamal" ، فإن مؤشر السجل سينتقل إلى نهاية الملف وذلك لعدم وجود اسم يطابق المطلوب. أما إذا قمت بالبحث الحساس ، فإن مؤشر السجل سينتقل إلى السجل الرابع "Kareem" وهو السجل المنطقي التالي بعد "Jamal" - من ناحية الترتيب الأبجدي - في قاعدة البيانات.

يعتبر البحث الحساس أكثر فائدة في بعض الأحيان من البحث العادي ، وخصوصا عندما تسمح للمستخدم أن يختار من قائمة السجلات.

ولإجراء البحث الحساس ، مرور العامل المنطقي (T.) كمتغير ثانٍ للوظيفة ( dbSeek ) كما تشاهد أدناه:

```
seek "LIEF"
dbSeek( "LIFE" )      // بالضبط مثل الأمر SEEK
dbSeek( "LIFE" , .t. ) // يقوم بإجراء البحث الحساس "softseek" //
```

### ملاحظة هامة

لاتشبه الوظيفة ( dbSeek ) بقية وظائف قاعدة البيانات الأخرى ، فإن هذه الوظيفة في الواقع ترجع قيمة قابلة للاستعمال: فإذا تمت عملية البحث بنجاح تكون القيمة الراجعة ( .T. ) ، أو ( .F. ) إذا فشلت عملية البحث.

### تحذير

تأكد جيدا من وجود فهرس نشط في منطقة العمل الحالية قبل القيام بعملية البحث.

## الأمر LOCATE <exp> / \_dbLocate(...)

يسمح لك هذا الأمر بالبحث بصورة متتابعة خلال بعض السجلات أو جميعها في قاعدة البيانات تحت شرط معين. ويعتبر هذا النوع من البحث أبطأ من SEEK ، ولكنه يمكن أن يكون مفيدا في الحالات التي تكون فيها معايير البحث معقدة التركيب ولا تتطابق ملف الفهرسة ( على سبيل المثال ، اسم العائلة = "الشهري" ويقوم في مدينة = "جدة" ). لا يلزم وجود ملف الفهرسة للقيام بعملية البحث هذه.

فإذا تم إيجاد البيانات المطلوبة، فإن علم الوظيفة ( FOUND ) سيضبط على True ، ومؤشر السجل سينتقل إلى السجل المناسب. أما إذا لم يتم إيجاد البيانات ، فإنه سيتم ضبط علم الوظيفة ( FOUND ) على False ، وعندها سيكون موقع مؤشر السجل يعتمد على نطاق البحث المعروف في Locate : فإذا كان نطاق البحث هو الملف بالكامل ، فإن موقع مؤشر السجل في هذه الحالة هو نهاية الملف ، وخلاف ذلك ، فإن موقع المؤشر سيكون عقب آخر سجل تم إجراء البحث عنه.

هل لاحظت الشرطة السفلية المزدوجة ( "\_\_" ) الموجودة في بداية الوظيفة `__dbLocate()` ؟ فهذه هي طريقة الشركة المنتجة لكليب في الإشارة إلى أن هذه الوظيفة تعتبر داخلية أو أنها قابلة للتغيير. ونحن لانحبذ استخدام مثل هذا النوع من الوظائف ، لأنها غير موثقة من الشركة وربما يتم تغييرها بشكل كبير (أو إلغاؤها كلياً) في الإصدار المستقبلي من كليب:

تشاهد أدناه مثلاً على أمر LOCATE:

```
locate for partnumber == "15224" .and. quantity > 100
```

في المثال التالي أدناه نعرض كيف يمكنك عمل بحث مشروط باستخدام LOCATE WHILE بالتزامن مع ملف الفهرسة النشط. إن هذه الطريقة أسرع من إجراء البحث LOCATE على اسم العائلة Lastname ، ثم بعد ذلك على الرصيد Balance.

```
seek "عسيري"
```

```
locate for balance > 200 while lastname == "عسيري"
```

## الأمر CONTINUE / `__dbContinue()`

يمكن أن يكون لكل منطقة عمل شرط LOCATE معلق خاص بها. ووظيفة الأمر CONTINUE هي السماح لك بمواصلة البحث LOCATE اعتماداً على الشرط الحالي لمنطقة العمل النشطة. وستبدأ عملية البحث اعتماداً على موقع السجل الحالي. فإذا تم إيجاد البيانات ، فإنه سيتم ضبط علم الوظيفة `FOUND()` على True ، وسينتقل مؤشر السجل إلى السجل المناسب. أما إذا لم يتم إيجاد البيانات ، فإن علم الوظيفة `FOUND()` (سيضبط على False ، وسيعتمد انتقال مؤشر السجل على نطاق البحث) كما هو مشروح أعلاه).



### تحذير

إذا استخدمت نطاق WHILE أو شرط مع LOCATE أو CONTINUE الأصلية ، فإنه سيتم تجاهلها. أما الشرط الوحيد الذي يستطيع الأمر CONTINUE النظر إليه فهو الشرط FOR. أما إذا كنت تحب مواصلة تعليق LOCATE الذي يتضمن الشرط WHILE ، فبدلاً من استخدام CONTINUE فإنه ينبغي القفز SKIP إلى الأمام إلى السجل التالي وكرر البحث LOCATE باستخدام الشرط REST.

## الأوامر والوظائف الأخرى لقاعدة البيانات

### تحذير

يتضمن كليب 5.2 على فاحص للخطأ بالنسبة للأوامر والوظائف التالية المرتبطة بقاعدة البيانات. لا تستخدمه دون أن يكون هناك قاعدة بيانات مفتوحة في منطقة العمل الحالية.

### الأمر dbAppend( ) / APPEND BLANK

عندما تريد إضافة سجل جديد لقاعدة البيانات ، يجب أولاً أن تضيف سجلاً فارغاً ثم تغير قيم الحقول حسب الحاجة. إن الأمر APPEND BLANK يضيف سجلاً فارغاً ، وسيكون موقع مؤشر السجل في هذه الحالة على السجل الجديد.

### أفكار عن الشبكة

إذا كانت قاعدة البيانات تحت الاستخدام المشترك في الشبكة ، فإن الأمر APPEND BLANK سيحاول قفل السجل الحالي. يمكنك استخدام الوظيفة ( ) NETERR لتحديد إذا فشلت عملية قفل السجل ، والتي تعني أن هناك مستخدماً آخر قام بقفل موقع السجل في الملف أو حاول إلحاق سجل جديد باستخدام APPEND BLANK في الوقت ذاته.

### الأمر COMMIT / dbCommitAll()

إن أية عملية تحرير تنفذ على قاعدة البيانات في كليبر يتم تخزينها في الذاكرة المؤقتة memory buffers. وتتم كتابة هذه البيانات بشكل منتظم إلى القرص ، ولكن ليس من الضروري أن يتم ذلك في كل مرة يتم فيها التعديل. وهذا يعني أنه من الممكن أن تفقد تعديلاتك بسبب انقطاع التيار الكهربائي أو لظرف من الظروف الأخرى. وبالإضافة إلى ذلك ، إن من الممكن أن يحدث تضارب إذا تم تنفيذ تطبيق كليبر على الشبكة ، وذلك لأنه قد يظل تعديل أحد الأشخاص في الذاكرة وبالتالي لن يظهر على محطة العمل الأخرى.

ولتجنب ضياع البيانات وحدوث هذا التنافر ، فإننا نقترح أن تستخدم الأمر COMMIT. حيث أنه يكتب محتويات الذاكرة المؤقتة buffer لكل مناطق العمل على القرص. وينبغي تدوين COMMIT بعد كل عملية تحديث على قاعدة البيانات.

### الأمر DELETE / dbDelete()

يقوم هذا الأمر بتعليم السجل الحالي لإجراء عملية الحذف. لاحظ أن هذا الأمر لا يقوم بعملية الحذف العضوية physical من قاعدة البيانات. إذ يوجد لكل سجل في قاعدة

البيانات علم حذف deleted flag مرتبط به. فعندما تقوم بتعليم السجل للحذف ، فانت بكل بساطة تقوم بتثبيت هذا العلم. أما الأمر الذي يقوم بالحذف الفعلي للسجل الذي تم تثبيت علم الحذف عليه ، فهو الأمر PACK (والذي سيناقتش فيما بعد).

### أفكار عن الشبكة

إذا كانت قاعدة البيانات تعمل على الشبكة بصورة مشتركة ، فإنه يجب قفل السجل الحالي باستخدام الوظيفة RLOCK ( ) و DBRLOCK ( ) قبل تعليمه من أجل الحذف باستخدام DELETE.

### الفهرسة INDEX ON / dbCreateIndex ( )

يستخدم هذا الأمر لإنشاء ملفات الفهرسة. تساند نسخة كليبر 5.2 الفهرسة المشروطة ، كذلك تسمح لك بأن تستخدم الأقواس المعقوفة داخل معالجة الفهرسة لتحديد الشروط. والتركيب اللغوي هو:

```
INDEX ON <expKey> TO <filename> [ UNIQUE ]  
      [ <SCOPE> ] [ FOR <ICondition> ]  
      [ WHILE <ICondition> ]  
      [ [ EVAL <ICondition> ] [ EVERY <nRecord> ] ]  
      [ ASCENDING | DESCENDING ]
```

إن الحد الأقصى لطول المتغير <expKey> هو ٢٥٠ حرفاً ، وهذا ينبغي أن يكون كافياً لكل التطبيقات. أما المتغير <filename> فهو يشير إلى اسم ملف الفهرسة المطلوب إنشاؤه. إذا لم تحدد امتداداً خاصاً بك ، فإن الملف سيأخذ الامتداد الافتراضي وهو ..NTX

أما الفقرة الاختيارية UNIQUE فهي تجبر ملف الفهرسة أن يحتوي على القيم الفريدة فقط. فعلى سبيل المثال ، إذا قمت بعمل الفهرسة بواسطة اسم العائلة lastname وكان

هناك عشرة أشخاص من عائلة " Al-maiman " فإن ملف الفهرسة سيأخذ أول واحد من هؤلاء العشرة.

تقوم العبارة التالية بإنشاء ملف الفهرسة CUSTOMER.NTX لفرز الأسماء بواسطة اسم العائلة lastname.

index on customer->lastname to customer

عندما تصدر الأمر INDEX ON ، فإن كل الفهارس الموجودة في منطقة العمل الحالية تقفل ويصبح ملف الفهرسة الجديد هو فهرس التحكم. أما مؤشر السجل فإنه سيقف على (في أعلى) أول سجل في ملف الفهرسة.

أما الخيار <scope> فهو يمثل جزءاً من قاعدة البيانات المطلوب فهرستها. والحالة الافتراضية لهذا الخيار هي "كل السجلات". ويمكنك تحديد السجل التالي <nRecord> أو الباقية REST (على سبيل المثال كل السجلات من الحالي إلى آخر سجل في قاعدة البيانات) ، أو سجل بعينه <nRec#>. إذا استخدمت المتغير <scope> مع الفهرس النشط ، فإنه يتم معالجة قاعدة البيانات اعتماداً على ترتيب فهرس التحكم الحالي. لاحظ أن المتغير <scope> مؤقت وبالتالي فإنه لن يخزن في ترويسة ملف الفهرسة ..NTX

أما الخيار FOR و الفقرة WHILE فهما امتداد رائع للفقرة <scope>. فيمكنك من خلال استخدام الخيار FOR إنشاء فهرس مشروط صحيح ، يحتوي على تلك السجلات التي تفي بالشرط <ICondition>. أما الفقرة WHILE فهي تقوم فقط بمعالجة السجلات ، ابتداءً من السجل الحالي الذي يحقق الشرط <ICondition>. وحالما يتوقف الشرط عن التحقق ، فإن عملية الفهرسة تنتهي. لاحظ أن الشرط FOR يخزن كجزء من ملف NTX. ويستخدم عند تحديث ملف الفهرسة. أما الشرط WHILE فهو مؤقت ويزول.

إن المثال أدناه يعرض استخدام كل من الشرطين FOR و WHILE.

```
// إنشاء فهرس مشروط لكل العملاء الذين عليهم رصيد مدين //
use customer
index on customer->lastname to temp for customer->balance > 0
```

```
// إنشاء فهرس فرعي لكل العملاء المقيمين في جدة وعليهم رصيد مدين //
use customer index city
seek "جدة"
index on customer->lastname to temp while city == "جدة"
for customer->balance > 0
```

```
// إنشاء فهرس فرعي لكل العملاء في جدة //
use customer index city
seek "جدة"
index on customer->lastname to temp while city == "جدة"
```

أما الفقرة EVAL فهي تسمح لك بتحديد الشرط الذي سيتم تقييمه أثناء عملية الفهرسة. وهذه الطريقة مثالية لتفادي تغذية المستخدم. إن الطريقة المثلى لاستخدام الشرط <ICondition> هي من خلال استدعاء وظيفة معرفة بواسطة المستخدم والتي تقوم بعرض معلومات عن الحالة وإرجاع القيمة المنطقية. أما إذا قام الشرط EVAL بإرجاع القيمة المنطقية (F.) فإن الفهرسة ستقف. إن استخدام الفقرة EVERY بالتزامن مع EVAL يمكنك من رفع مستوى الأداء بواسطة تقييم الشرط فقط في فترة الضبط السبقي وليس لكل سجل.

يعرض المثال التالي كيف يمكنك مشاهدة حالة عملية الفهرسة ، والتي تجعل المستخدم يشعر بالأمان.

```
use customer new
index on customer->lastname to lastname eval Progress( )
return nil
```

```
function progress
@ 24, 00 say "indexing process: " + ;
```



```
str(recno( ) / lastrec( ) * 100, 6, 2) + ;  
"% complete "
```

```
return .t.
```

أما فقرات الفرز الاختيارية ASCENDING و DESCENDING فهي تجبر مفاتيح الفهرسة أن تفرز بالترتيب التصاعدي ASCENDING أو التنازلي DESCENDING على التوالي. الوضع الافتراضي في حالة عدم تعيين طريقة الفرز ، هو الفرز التصاعدي ASCENDING. ويفضل في الفقرة DESCENDING استخدام الوظيفة DESCEND ( وذلك لأنها أسرع بكثير.

### فكرة

إذا كنت تريد إنشاء فهرس ألفبائي للأسماء الانجليزية فإنه ينبغي استخدام الوظيفة UPPER( ) . خلاف ذلك ، فإن كل المدخلات بالحروف الكبيرة upper-case ستظهر قبل المدخلات بالحروف الصغيرة lower-case. وهذا ناتج عن كون الحروف الصغير مفصولة عن الحروف الكبيرة في جدول الآسكي ASCII.

### تحذير

ذكرنا أثناء مناقشة حقول المذكرة Memofields ، أن مفاتيح الفهرسة في كليبر يجب أن تكون ثابتة الطول. وبناء على ذلك ، إذا استخدمت الوظيفة TRIM( ) فأنت تبحث عن المتاعب ، اللهم إلا إذا استخدمت أيضا الوظيفة PADR( ) لإجبار كل مفتاح فهرسي أن يكون بالطول ذاته.

## الأمْر PACK / dbPack( )

يقوم هذا الأمر بإزالة أية سجلات محذوفة من قاعدة البيانات في منطقة العمل الحالية (انظر أمر DELETE أعلاه).

### فكرة عن الشبكة

في هذه الحالة يجب أن تكون قاعدة البيانات خاصة exclusive (ليست SHARED) لكي تستطيع إزالة هذا السجل (أو السجلات).

## الأمْر RECALL / dbRecall( )

يقوم هذا الأمر بإزالة محدد الحذف من السجل الحالي (انظر الأمر DELETE أعلاه).

### فكرة عن الشبكة

إذا كانت قاعدة البيانات مشتركة ، يجب أن تقوم بإقفال السجل الحالي باستخدام الوظيفة ( ) RLOCK قبل أن تحاول عمل RECALL للسجل.

## الأمْر REINDEX / dbReindex( )

يقوم هذا الأمر بإعادة بناء كل السجلات النشطة في منطقة العمل الحالية. ولا يفضل استخدام الأمر REINDEX للأسباب التالية : (١) لأنه يجب إيقاف الاستخدام المشترك لقاعدة البيانات (٢) لايقوم بإعادة إنشاء ترويسة ملف الفهرسة ، وبناء على ذلك فإنه لايقوم بتصحيح تحريف ملف الفهرسة. ونحن نقترح استخدام الأمر INDEX ON لإعادة إنشاء الفهارس.

## الأمر dbSelectArea(<n>) / SELECT <n>

يسمح لك هذا الأمر بإنشاء منطقة عمل نشطة أخرى. يمكنك إما أن تحدد رقماً (وهذا غير مستحسن) أو الحرف المستعار characteralias. فعلى سبيل المثال، نجد أن العبارة statement التالية تجعل منطقة العملاء نشطة:

```
select customer
```

ولأنشجع من استخدام الأمر SELECT بدلا من عامل الاستعارة ، والذي سنناقشه فيما بعد.

## الأمر dbSetFilter(<expr>) / SET FILTER TO <expr>

إن هذا الأمر يسمح لك بمعالجة مجموعة فرعية من السجلات في منطقة العمل الحالية اعتمادا على شرط محدد. إن هذا الأمر قوي جدا ، ولكن قد يكون بطيئاً بشكل غير عادي بالنسبة للملفات الكبيرة. وعندما تقوم بضبط أمر الترشيح FILTER ، يجب أن تنقل مؤشر السجل لتجهيزه. إن الاستخدام الذي ننصح به هو SET FILTER ثم GO TOP لتجهيز الترشيح.

عندما يتم ضبط الترشيح ، فإن المرشح يتجاهل السجلات التي لا تطابق شرط الترشيح. إن الاستثناء الوحيد هو جملة GO والتي تستمر في إجبار مؤشر السجل على تحديد رقم السجل سواء كان صحيحا أم لا.

إذا تم ضبط الترشيح filter عند إنشاء الفهرس ، فإن كل السجلات مع ذلك ستعكس في ملف الفهرس (ليس المرئي منها فحسب).

## الأمر dbClearFilter( ) / SET FILTER TO

يقوم هذا الأمر بمسح شرط الترشيح الموجود في منطقة العمل الحالية.

## الأمر dbSetIndex(<n>) [ ADDITIVE ] / SET INDEX TO <n>

إن هذا الأمر يسمح لك بإنشاء فهرس نشط. يمكنك تحديد اسم ١٥ ملفاً فهرسة كأقصى حد مع الأمر SET INDEX TO ، والذي يعادل بين واحد وخمسة عشر استدعاء للوظيفة dbSetIndex( ) . إذا استخدمت الوظيفة dbSetIndex( ) لتنشيط الفهرس ، فإن كل الفهارس الأخرى في منطقة العمل الحالية ستظل نشطة. كما يمكنك استخدام فقرة الخيار ADDITIVE والتي تقوم بأداء الوظيفة ذاتها.

### تنبيه

إذا كان لديك أكثر من ملف فهرس لقاعدة بيانات معينة ، فإنه ينبغي عليك في هذه الحالة التأكد من أنها كلها نشطة عندما تقوم بعمل أي تحديث على قاعدة البيانات. خلاف ذلك ، فأنت تعرض ملفات الفهرسة لخطر التلف ، والذي سينتج عنه تحطيم برنامجك.

## الأمر dbClearindex( ) / SET INDEX TO

يقوم هذا الأمر باغلاق كل ملفات الفهرسة النشطة في منطقة العمل الحالية.

## الأمر dbSetOrder(<n>) / SET ORDER TO <n>

يسمح لك هذا الأمر بتحديد الفهارس النشطة في منطقة العمل الحالية التي ستضبط عملية ترتيب الفرز. إن الفهارس مرقمة ابتداءً من ١ ، اعتماداً على ترتيب فتحها. فعلى سبيل المثال ، إن العبارات التالية ستجعل فهرس الشركة يتحكم في الترتيب:

```
use customer
set index to lastname, city , company
set order to 3
```

## الأمر dbSetRelation( ) / SET RELATION TO...INTO...

يقوم هذا الأمر بربط قاعدة بيانات بأخرى اعتماداً على مفتاح ( أو رابط ) key expression. وكل منطقة عمل " أم " يمكن أن تحتوي على ثمان مناطق عمل " أبناء " مرتبطة بها كأقصى حد. وعندما يتم ضبط العلاقة بين قواعد البيانات ، فإن تحرك أي مؤشر سجل في منطقة العمل " الأم " يجبر مؤشر السجل في منطقة العمل " الابن " على الحركة أيضاً.

ولكي تضبط العلاقة داخل منطقة الابن ، ينبغي أن يكون لمنطقة الابن فهرس نشط يطابق تعبير الربط. وكلما قمت بتحريك مؤشر السجل في منطقة العمل الأم ، فإن عملية البحث SEEK تتم في منطقة العمل الابن. فإذا لم يوجد ما يطابق في منطقة العمل الابن ، فإن مؤشر السجل سيقف في نهاية الملف ، وسيتم ضبط علم الوظيفة ( EOF ) على True.

تضبط العبارات التالية العلاقة بين قاعدة بيانات الفواتير INVOICES و LINEITEM. فعندما تقوم بالبحث عن الفاتورة رقم ١٢٣٤٥ في منطقة عمل قاعدة البيانات INVOICES ، فإن مؤشر السجل في منطقة عمل قاعدة البيانات LINEITEM أيضاً سيقف على أول سجل يطابق رقم تلك الفاتورة.

```
use lineitem index lineitem new
use invoices index invoices new
```



```
set relation to inv_no into lineitem  
seek "12345"
```

### الأمر dbClearRel( ) / SET RELATION TO

يقوم هذا الأمر بتحرير كل العلاقات المعرفة في منطقة العمل.

### الأمر dbUnlock( ) / UNLOCK

يستخدم هذا الأمر لإطلاق الملف/السجل المقفل في منطقة العمل الحالية والتي تم ضبطها بواسطة المستخدم الحالي. انظر العنوان " نظام الشبكة " في الصفحة التالية لمزيد من التفاصيل.

### الأمر dbUnlockAll( ) / UNLOCK ALL

يستخدم هذا الأمر لإطلاق ملف/سجل مقفل في كل مناطق العمل الحالية والتي ضبطها المستخدم. انظر العنوان " نظام الشبكة " في الصفحة التالية لمزيد من التفاصيل.

### الأمر dbZap( ) / ZAP

يقوم هذا الأمر بإزالة كل السجلات بصورة عضوية من قاعدة البيانات في منطقة العمل الحالية ، وسواء أتم تعليمه بعلامة الحذف أم لا. لا تحاول الإكثار من استخدام هذا الأمر دون داع.

---

#### فكرة عن الشبكة

يجب أن تكون قاعدة البيانات خاصة أو حصرية EXCLUSIVE ، لكي تستطيع استخدام الأمر ZAP.



## منهج ونظام الشبكة

إن تطبيقات كليب على وجه العموم جاهزة للعمل على الشبكة ، إذ أن التطبيقات تفتح قواعد البيانات جميعها للعمل المشترك. ولعلك قد لاحظت لاحظت أننا كررنا أكثر من مرة أثناء الشرح والمناقشة ، وبالتحديد عند أي تحديث لقواعد البيانات فإن هذا يتطلب أن يكون السجل مقفلاً أو يكون الملف مقفلاً أو استخدام الأمر EXCLUSIVE.

### الأمر SET EXCLUSIVE

ينبغي استخدام الأمر EXCLUSIVE مع قاعدة البيانات إذا كنت تريد استخدام أحد الأوامر PACK ، أو ZAP ، أو REINDEX. ويمكنك إجبار قواعد البيانات جميعها للعمل المشترك على الشبكة بإصدار الأمر SET EXCLUSIVE OFF في بداية برنامجك ( الوضع الافتراضي هو أن كل الملفات ستفتح للعمل المفرد).

لاحظ أنه لا يهم ما عليه وضع الضبط العام لـ: EXCLUSIVE ، فإنه لا زال بإمكانك إجبار قاعدة البيانات أن تفتح للعمل المشترك أو الفردي بواسطة استخدام فقرات الخيار SHARED و EXCLUSIVE بالتزامن مع الأمر USE.

### الوظيفة RLOCK( )

تستخدم هذه الوظيفة على وجه العموم في القيام بتحديث سجل مفرد. فهي تحاول قفل السجل الحالي في منطقة العمل الحالية. إن الأوامر التي تتطلب قفل السجل هي : REPLACE و DELETE و RECALL ، و جملة التعيين السطري الخاصة بتعيين القيم لحقول قاعدة البيانات. تقوم الوظيفة RLOCK( ) بإرجاع القيمة المنطقية True إذا تمت

عملية قفل السجل بنجاح ، أو القيمة False فشلت عملية قفل السجل. إذا تمت عملية ( ) RLOCK بنجاح ، فإنه سيستمر قفل السجل إلى أن تستخدم الأمر UNLOCK أو UNLOCK ALL أو تغلق قاعدة البيانات أو تقوم بقفل سجل آخر باستخدام الوظيفة ( ) RLOCK أو تقوم بقفل الملف الحالي باستخدام الوظيفة FLOCK().

لاحظ أن المستخدمين الآخرين لازال بإمكانهم القراءة فقط من قاعدة البيانات الحالية حتى إذا قمت بإغلاق السجل (علماً بأنك قد تكون الشخص الوحيد الذي سيكون بمقدوره تعديله).

## الوظيفة ( ) FLOCK

تستخدم هذه الوظيفة بشكل عام عندما ترغب في القيام بتحديث سجلات متعددة في قاعدة البيانات. إن الأوامر التي تستدعي إغلاق الملف هي : APPEND FROM و DELETE و REPLACE وأخيراً RECALL من أجل السجلات المتعددة و UPDATE. إذا تمت عملية إغلاق الملف بنجاح ، فإن الملف سيظل مغلقاً إلى أن تستخدم الأمر UNLOCK أو UNLOCK ALL أو إغلاق قاعدة البيانات أو إصدار أمر بقفل السجل بواسطة الوظيفة ( ) RLOCK.

## استراتيجيات قفل السجل

نعود فنقول ، عندما ترغب في تحديث سجل في قاعدة البيانات ، فإنه يجب قفل هذا السجل عن بقية المستخدمين على الشبكة. إن هناك اثنين من الاستراتيجيات الشائعة لإغلاق السجل. وتجنباً للطريقة التقليدية في الجزم بأن واحدة منهما أفضل من الأخرى فإننا سنقوم بمناقشة موجزة لكل إستراتيجية ، ثم نترك لك استنتاج الحكم النهائي عليها.

## تحرير - قفل Lock - كتابة - إزالة القفل Unlock

لا يتم إصدار الوظيفة ( RLOCK ) إلا قبل عملية كتابة التعديل الفعلي مباشرةً. وهذا يعتبر جيداً لأنه يسمح لك بقفل السجل أقل مدة ممكنة من الوقت. ومن ناحية أخرى فإن هذه الطريقة يمكن أن تتسبب في خلط ناتج عن أنها تسمح بإمكانية أن يقوم اثنان من المستخدمين بتعديل السجل ذاته في الوقت ذاته وبناء على ذلك فإن هذا يعني أن كل واحد يطمس تعديل الآخر.

## قفل Lock - تحرير - كتابة - إزالة القفل Unlock

وفي هذا السيناريو ، يتم استدعاء الوظيفة ( RLOCK ) قبل أي تحرير Edit. وهذا يعتبر أمراً جيداً ، فهو يتحاشى إمكانية قيام مستخدمين بتحرير السجل ذاته. من ناحية أخرى ، إلا أنه قد يسبب إلى حد ما مشكلة بسيطة وهي أن المستخدم يمكنه السير بعيداً عن محطة العمل وترك السجل مقفلاً (وبناء على ذلك يصبح السجل غير متوفر للمستخدمين الآخرين).





## أنواع البيانات Data Types

يوفر كليب ثمانية أنواع من البيانات. لاحظ أن المتغيرات في كليب هي من النوع الحر ، وهذا يعني أنه يمكننا تغيير نوع بياناته " في الهواء " .

لقد ناقشنا منها الحرفي character ، و التاريخي Date ، والرقمي Numeric ، والمنطقي Logical في قسم نوع حقل قاعدة البيانات. وهذا ينطبق على المتغيرات كما ينطبق الحقول. أما الأنواع الأربعة الباقية فهي المنظومات arrays ، القيمة العديمة Nil ، وكتل الشيفرة Code Blocks ، ولعلك تستخدم "العناصر" فهي أفضل Objects.

## المنظومات Arrays

المنظومات هي أحد أنواع البيانات في كليب ، والتي تحتوي على مجموعة من البيانات الأخرى. وتخزن هذه البيانات في " عناصر " elements. يمكن أن تتكون هذه المجموعة من ٤٠٩٦ عنصرا كحد أقصى على سبيل المثال. كما أن كليب يتميز عن بقية لغات البرمجة بأنه يسمح بخلط أنواع البيانات داخل المنظومة. وهذا قطعاً يعطي المنظومة قوة رائعة ، ويسمح لك بمحاكاة بنية البيانات في لغة سي.

الأمر الممتع حقاً في استخدام المنظومات بدلا من عدد من المتغيرات هو أن كل العناصر في المنظومة متجمعة معا بصورة منطقية. وهذا طبعا يساعدك كثيرا ويسهل عليك معالجتها. وعلى العكس من ذلك فقد تواجه الكثير من المصاعب عند العمل مع متغيرات مختلفة.

يمكن تغيير حجم منظومة كليب ديناميكيا بواسطة كل من الوظيفتين ( AADD و ( ASIZE. كما يمكن أن تتداخل ، مثلا ، يمكن أن يحتوي عنصر المنظومة على إسناد

لنظومة أخرى. إن هذين العاملين ، يوفران إمكانية خلط أنواع البيانات داخل المنظومة ذاتها ، وهذا يجعل مطوري كليبر يحققون كل مايطمحون إليه عند استخدام المنظومات.

## العدم NIL

إن الاستخدام الهام لنوع البيانات NIL هو كمتغير لفحص وظائفك. فعندما تقوم بإعلان المتغيرات كالمحلي LOCAL أو الساكن STATIC أو الخاص PRIVATE يتم استهلاكها ضمناً بالقيمة NIL. وهذا يساعدنا على مشاهدة هل تم استقبال القيمة فعلياً كمتغير أم لا. سنشاهد أمثلة على هذا الموضوع عندما نناقش قضية كتابة الوظائف التي يحددها المستخدم user-defined functions. ويختصر هذا الاسم في العادة ليكون UDF's.

أما الاستخدام الآخر لـ NIL فهو عندما تكتب وظيفة يحددها المستخدم تقوم بإرجاع قيمة لاعلاقية. وهذا يكافئ " void function " في لغة سي. ومن خلال تعيين NIL كقيمة راجعة ، يمكنك من النظرة الأولى معرفة ما إذا كانت القيمة الراجعة لها أية قيمة هامة. وهذا في الواقع يمكن أن يوفر لك الوقت الذي يضاع عادة في صيانة البرامج.

## كتل الشيفرة Code Blocks

إن كتل الشيفرة هي نوع خاص من أنواع البيانات والتي تحتوي على شفرة كليبر مسبقة الترجمة. وبهذا بالمعنى يمكنك اعتبارها كوظيفة function. إلا أن كتل الشيفرة تختلف بميزة واحدة عن الوظائف التقليدية وهي : أنه يمكن تمريرها كمتغيرات parameters للوظائف الأخرى.

البنية الهيكلية لكتلة الشيفرة هي :

```
{ | [ <argument list> ] | <expression list> }
```

تبدأ كتل الشيفرة بفتح المعقوفة ( "{ " ) وتنتهي بإغلاقها ( " } " ). وتحتوي على رمز الأنبوب pipe ( " | " ) مباشرة بعد فتح المعقوفة. وهذا الأنبوب هو محدد اختياري لقائمة المتغيرات <argument list> ، والتي ينبغي أن تمرر لكتلة الشيفرة عند التقييم. كما ينبغي أن يفصل بين هذه المتغيرات داخل القائمة <argument list> بالفاصلة (على سبيل المثال " a, b, c...." ).

أما <expression list> فهي قائمة من أية تعبيرات شرعية في كليبر ، يتم الفصل بين كل تعبير وآخر بالفاصلة. وهذا يمكن أن ينفذ سلم النغم ، كما ستكتشف سريعا. في الواقع ، الشيء الوحيد الذي لا يمكنك استخدامه في كتل الشيفرة هو أوامر كليبر ، مثل الأمر SEEK. ، وكما شاهدت سابقا ، فإن معظم أوامر كليبر الهامة ، توجد لها وظائف تكافئها في العمل في كليبر. فبدلا من استخدام الأوامر في كتلة الشيفرة ، يمكنك بكل بساطة استدعاء الوظيفة.

ومع أننا سنقوم بعد هذا الشرح الموجز بالدخول في أعماق كتل الشيفرة تفصيلا. إلا أننا سنلقي نظرة سريعة على نوع العلاقة بين الوظيفة وكتلة الشيفرة.

```
b := { | | 50 }
x := eval(b, 200)
? x
? myfunc(2000)
return
```

```
function myfunc(x)
return x * 50
```

ستقبل الوظيفة ( ) MyFunc متغيراً واحداً (X) وترجع قيمته مضروبا بـ: ٥٠. وهذا أيضا ما ستفعله كتلة الشيفرة B. إن الوظيفة ( ) EVAL الموجودة في كليبر تستخدم لـ "تقييم" كتلة الشيفرة code block والتي هي أساس كاستدعاء للوظيفة. لاحظ أن الرقم ٢٠٠ مرر كمعلم ثانٍ للوظيفة ( ) EVAL وأي شيء وراء المعلم الأول سيمرر مباشرة حتى النهاية إلى كتلة الشيفرة.

كما يمكن تقييم كتلة الشيفرة باستخدام الوظائف ( ) EVAL ، و ( ) AEVAL ، و ( ) DBEVAL. كما أن هذه الوظائف يتم تقييمها داخليا بواسطة وظائف أخرى معينة مثل ( ) ASCAN ، و ( ) ASORT. تقوم كتلة الشيفرة بإرجاع قيمة التعبير expression الموجود في أقصى اليمين.

ولمزيد من التفاصيل عن كتل الشيفرة code blocks وكيف تستخدمها لتجني منافعها ، فإننا ننصحك بقراءة المذكرة الثانية من هذه المحاضرات والموسومة بالعنوان " أساسيات كليبر 5.2 " .

## الأهداف Objects

قبل أن نناقش الأهداف Objects ، يجب علينا أولا الحديث عن فئة: Class. فئة: Class هي تعريف أو قالب. فهي لا تؤدي أي عمل بذاتها. ولذلك فإننا نستخدم هذا القالب لإنشاء فوري للفئة class ، والتي تعرف بـ: Objects. والتي تعيش وتنفس في أعماق برنامجك.

إن الهدف Object هو مجموعة من البيانات والرميز. يتم حفظ البيانات في متغيرات فورية instance variables ، والرميز يمثل بواسطة الوظائف (النظم) التي تعمل على تلك البيانات. وهذا يعني أنه يمكن القيام بتعديل البيانات ومشاهدتها فقط بواسطة الوظائف التي تحتاج للعمل عليها.

وعلى الرغم من أن المناقشة المطولة عن الكائنات Objects بعيدة عن مجال هذه المذكرة إلا أن هناك فائدتين أساسيتين من استخدام object-oriented وهي الكبسلة encapsulation ، و القدرة على تكرار استخدام البرامج من خلال الوراثة inheritance.



## الكبسلة Encapsulation

الكبسلة هي إخفاء البيانات عن فئة class ، بحيث يمكن الوصول إليها فقط من خلال طرقها methods. وهذا يجعل الفئة تعمل بالضبط كالصندوق الأسود ، فهي توفر على المبرمجين الآخرين صعوبات اكتشاف أسماء المتغيرات المتضاربة.

## الوراثة Inheritance

هي إحدى الخصائص التي تتميز بها OOP عن البرمجة التقليدية. فعندما نشير إلى البرامج القابلة للاستخدام المتكرر ، فإننا نعني بشكل عام مكتبات الوظائف. ولكن دعنا نفترض أنك أنشأت مكتبة وظائف خاصة بك ، تحتوي على الوظيفة ( XYZ ). وفي مرة ثانية وجدت أن الوظيفة ( XYZ ) مناسبة للاستخدام مرة أخرى تقريبا ، ماعدا أنها تحتاج إلى بعض الإضافات. حينها ينبغي عليك أن تقوم بالقص واللصق شيفرات الوظيفة ( XYZ ) في وظيفة جديدة ، ثم تضيف عليها الشيفرات الجديدة المطلوبة لجعلها تعمل بالضبط كما هو مطلوب.

أما مع OOP فإنه ينبغي أن لا تحتاج مطلقا لعملية القص واللصق. فكل فئة كائنات object class لها القدرة على أن تكون الفئة الأعلى superclass ، وبناء على ذلك تمر سماتها attributes إلى أسفل للابن child أو الفئة الفرعية subclass. وعند ذلك يمكن للفئة الفرعية أن تضيف المتغيرات الفورية instance variables / methods أو أن تبدل الموجودة. في المثال أعلاه ، ينبغي أن يكون لدينا فئة كائن XYZ ، ثم ننشئ فئة فرعية نسميها WXYZ وهي الوراثة من XYZ. والتي ينبغي أن تظهر بالسلوك والسمات ذاتها لـ: XYZ ، وينبغي عليك أن تضيف عليها فقط الأجزاء غير المتوفرة في XYZ.

## طبقات الهدف في كليبر 5.2

هناك العديد من طبقات الكائن السابقة التعريف في كليبر 5.2:

– طبقة TBROWSE.

– طبقة TBCOLUMN.

– طبقة GET.

– طبقة ERROR.

ولمزيد من المعلومات عن هذا الموضوع الشائق ، يمكنك الرجوع إلى المذكرة المعنونة بـ: ” طبقات الكائنات في كليبر 5.2 “. في النسخة الحالية من كليبر، من غير الممكن أن تنشئ طبقات خاصة بك. إلا أن الشركة المنتجة لكليبر تعد أن يكون هناك مساندة كاملة لهذا الموضوع في الإصدار القادم. وفي الوقت الحالي فإن العديد من المطورين (بما فيهم مؤلف الكتاب) يستخدمون إما Class(y) (من شركة ObjectTech) أو SuperClass (من شركة ChyDale Software) لإنشاء object class خاصة بهم و تورث من طبقات كليبر الافتراضية.

## اختبار أنواع البيانات Testing Data Type

ستحتاج في الغالب إلى تحديد نوع بعض البيانات لكي تتمكن من معالجتها. إن أفضل طريقة لعمل ذلك هي استخدام الوظيفة VALTYPE( ) والتي تقبل البيانات وترجع حرفاً من الحروف المذكورة أدناه والتي تعرف بنوع البيانات:

النوع	الحرف
منظومة Array	A
كتلة مشفرة Code block	B
حرف Character	C
تاريخ Date	D
منطق Logical	L
رقم Numeric	N
هدف Object	O
عديم NIL	U

فعلى سبيل المثال ، VALTYPE("Helo") فإنها سترجع "C" و VALTYPE("456") سترجع "N".





## العوامل Operators

تزوّدنا كليبّر 5.2 بمجموعة عوامل قوية. وهي تقسم إلى خمسة أصناف عامة:

- الرياضيّة Mathematical
- العلائقية Relational
- المنطقية Logical
- التعيين Assignment
- الأغراض الخاصّة Special Purpose

## العوامل الرياضيّة Mathematical Operators

من الضروري جداً أن يكون المبرمج لأي لغة على مدركاً تماماً ما يخص العوامل الرياضيّة وأسبقيتها. وتحتوي كليبّر 5.2 على ثمانية عوامل من هذا النوع. اثنان منها ( وهما "+" و "-" ) يمكن أن تؤدي أكثر من عملية ، ويمكن استخدامها أيضاً في التاريخ و التعبيرات الحرفيّة.

العامل	الوظيفة	أنواع البيانات الممكنة
<sup>^</sup> , **	الأس Exponentiation	رقمي N
*	الضرب Multiplication	رقمي N
/	القسمة Division	رقمي N
%	الباقى Modulus	رقمي N
+	الجمع ، موجب أحادي	رقمي N و تاريخ D
+	الربط Concatenate	حرفي C



الجدول مستمر من الصفحة السابقة.....

العامل	الوظيفة	أنواع البيانات الممكنة
-	طرح ، سالب أحادي	رقمي N ، تاريخ D
-	الربط Concatenate	حرفي C
++	تزايد Increment	رقمي N ، تاريخ D
--	تناقص Decrement	رقمي N ، تاريخ D

## الأسس Exponentiation

عامل الأسس يرفع التعبير الرقمي الأول إلى قوة الثاني. إن رفع العدد إلى القوة المعطاة يشبه من الناحية العامة عملية ضرب العدد بنفسه عدداً من المرات. فعلى سبيل المثال ، رفع العدد ٣ للقوة الثالثة يعني أن يصبح العدد ٢٧ ( $3 \times 3 \times 3$ ). إن أي رقم يتم رفعه إلى قوة بالسالب فإن الناتج يساوي مقلوب القوة الموجبة. فعلى سبيل المثال ، ٢ ترفع إلى القوة -٢ ، لأنها ستساوي ٢٥. ( $(2 \times 2) / 1$ ).

لاحظ أن "\*" و "^" متكافئان من الناحية الوظيفية.

```
? 3 ** 2 // 9
? 10 ** 0 // 1
? 10 ** 2 // 100
? 10 ** -2 // .01
```

## الباقى Modulus (remainder)

إن عامل الباقي Modulus هو قسمة تعبير رقمي على آخر وإرجاع باقي القسمة. فإذا جعلت المقسوم عليه صفراً ، فإنك ستحصل في هذه الحالة على رسالة خطأ قاتل من المترجم fatal compiler error.

إذا لم يسبق لك استخدام عامل الباقي ، فانظر إلى المثال أدناه ، لكي تعرف كيف تستخدم هذا العامل في برامجك لإنجاز عمليات مشابهة لها:

```
? 100 % 10      // 0
? 5 % 6          // 5
? 4 % 3          // 1
```

## الوظيفة MOD ( )

إن الوظيفة MOD ( ) هي من وظائف dBASE III PLUS. إن توفر هذه الوظيفة في كليبر هو فقط لأغراض التوافقية ، خاصة في حالة وجود برامج مكتوبة بـ: dBASE III PLUS ويراد عمل ترجمة لها باستخدام كليبر دون القيام بأي تعديل يذكر. وإلا فإنه تم في كليبر استبدال هذه الوظيفة بعامل الباقي % وهو كما ترى أسهل في الكتابة وأسرع في التذكر من الوظيفة MOD ( ) التي لا نحبذ لك استخدامها.

إن الجدول التالي أدناه يشرح الفروقات بين وظيفة MOD ( ) الموجودة في dBASE III PLUS وعامل الباقي % في كليبر :

المقسوم	المقسوم عليه	نتيجة	استخدام	النظام
		عامل الباقي %	كليبر MOD()	dBASE MOD()
3	0	خطأ مجمع	0*	3
3	-2	1	-1	-1
-3	2	-1	1	1
-3	0	خطأ مجمع	0*	-3
-2	3	-2	1	1

\* في الواقع ، تنشئ هذه خطأ في زمن التنفيذ ، ولكن نظام التعامل مع الأخطاء الموجود في كليبر 5.2 يحدد أخطاء القسمة على صفر وتقوم بإصلاح ذلك بإرجاع صفر للجملية التي صدر منها الخطأ. وقد لا ترغب بأن يحدث هذا بصورة دائمة.

## العامل " + "

يمكن أن يأخذ هذا العامل أحد ثلاثة أشكال:

- الجمع Addition: يمكن استعماله إما لجمع تعبيرين رقميين أو تاريخ ورقم.
- الموجب المستقل Unary positive: وهو أن يسبق العدد بالإشارة "+", وهذا يجعل الرقم يأخذ المستوى الأعلى في الأسبقية على العمليات الرقمية الأخرى (ماعد السالب المستقل).
- التسلسل Concatenation: وهو ربط تعبيرين حرفيين (سواء أكانتا سلسلتين Strings أو مذكرتين Memos) معا.

وكما تشاهد في المثال، فإن أنواع البيانات الخاصة بالمعاملات operands التي تقوم بتعيينها هي التي تحدد نوع العملية التي سيقوم العامل " + " بإجرائها على الأرقام والمتغيرات (المعاملات).

```
/* addition */
? 5 + 100           // 105
? 31 + ctod("12/01/94") // 01/01/95
/* concatenation */
mstring := "CA-Clipper"
? mstring + " 5.2 " // CA=Clipper 5.2
```

### ملاحظة

وبما أننا نتحدث عن موضوع العامل Operator والمعامل Operand وقد يحدث بينهما كثير من اللبس على القارئ. فإن الفرق الواضح بينهما هو أن العامل Operator هو مجموعة الرموز والأرقام التي تستخدم لتعيين العمليات الرياضية مثل العوامل الرياضية + و - و % وغير ذلك. أما المعامل Operand فهو الكمية المشغلة حسابيا لتلك المعاملات السابقة ، فالمعامل هو القيم والمتغيرات التي تجري عليها العمليات الحسابية باستخدام العوامل للحصول على النتائج.

### العامل “ - ”

وكما هو الحال بالنسبة للعامل “ + ” ، فإن العامل “ - ” يمكن أن يعني أحد ثلاثة أشياء:

- الطرح Subtraction: يمكن استعماله إما لطرح تعبيرين رقميين أو تاريخين أو تاريخ ورقم. أما الوضع المقلوب فهو غير مقبول بالنسبة لطرح التاريخ من الرقم.
- السالب المستقل Unary negative: هو أن يسبق العدد بالإشارة “-” ، والتي تؤثر في ضربه بـ: -1 . وهذا العملية تأخذ الأسبقية على كل العمليات الرقمية الأخرى ماعدا الموجب المستقل.
- التسلسل Concatenation: هو ربط تعبيرين حرفيين ( سواء أكانا سلسلتين Strings أو مذكرتين Memos ) معا. كما أن هذا العمل يتميز عن عامل الموجب المستقل ، بأنه يقوم بإزالة المسافات التابعة للتعبير الأول ويحولها إلى داخل نهاية السلسلة الراجعة.

كما أن هذا العامل “-” يقوم تلقائيا بتحديد العملية التي سيقوم بإنجازها اعتمادا على أنواع البيانات الخاصة بالمعاملات Operands المستخدمة معه.

```

/* subtraction */
? 25 - 21 // 4
? 31 - ctod("12/01/90") // خطأ زمن التنفيذ
? ctod("01/01/91") - 31 // 12/01/90

/* unary negative */
mnum = 500
? - mnum // -500
? 250 - -mnum // 750

/* cncatenation */
mstring = " CA-Clipper "
? mstring - "5.2" // " CA-Clipper5.2 "
? "I love" - mstring + "5.2" // "I love CA-Clipper 5.2"

```

## عاملا التزايد والتناقص

لقد استخدم مبرمجو لغة سي هذين العاملين منذ سنوات طويلة. فاحدهما يزيد والآخر ينقص قيمة المتغير بواحد. ويمكن تطبيق هذين العاملين على متغيرات الأرقام والتاريخ أو حتى حقول قواعد البيانات.

### ملاحظة

إذا استخدمت هذين العاملين بالتزامن مع حقل ، فإنه يجب عليك أن تستهل الحقل باسم قاعدة البيانات الرمزي alias.

وهناك طريقتان لاستخدام عاملي التزايد والتناقص. فإنه يمكنك استخدامهما إما قبل المتغير أو بعده ("prefix") أو بعد ("postfix"). فإذا استخدمت الشكل prefix ، فإن المتغير سيتزايد أو يتناقص قبل أن تستخدم قيمته في مكان آخر. أما الشكل postfix فيجعل عملية التزايد أو التناقص تتأخر إلى ما بعد تقييم باقي التعبيرات.

تأكد من أنك قد استوعبت كل الفروق بين prefix و postfix قبل الشروع في البرمجة باستخدامهما.



وهناك مثال آخر يوضح الفروق بين هذين العاملين:

```
yy := 5
zz := -- yy           // prefix
? yy                  // 4
? zz                  // 4
zz := yy++            // postfix
? yy                  // 5
? zz                  // 4
zz := yy + yy++       // postfix
? yy                  // 6
? zz                  // 10
zz := --yy + yy       // prefix
? yy                  // 5
? zz                  // 10
zz := yy + --yy       // prefix
? yy                  // 4
? --zz                // 8
? zz                  // 8
```

## أسبقية العوامل الرياضية

يحسن بنا هنا أن نذكر ترتيب أسبقية العوامل الرياضية في كليب 5.2 :

١ - الموجب والسالب (+ ، -)

٢ - التزايد القبلي والتناقص القبلي ( ++ ، -- )

٣ - الرفع لقوة الأس ( ^ ، \*\* ) .

٤ - الضرب والتقسيم ومعامل الباقي ( % ، / ، \* )

٥ - الجمع والطرح ( + ، - )

٦ - لاحقة الزيادة والإنقاص ( ++ ، -- )

عندما نستخدم أكثر من معامل رياضي واحد ، فإن مترجم كليب يقيم كل " عبارة ثانوية " في مستوى الأولوية قبل العبارات الثانوية في المستوى التالي . وتنفذ جميع العمليات بالترتيب من اليسار إلى اليمين .

يمكننا استخدام الأقواس لتجاوز هذا الترتيب الافتراضي . وتقوم أية " عبارة ثانوية " ضمن قوسين أولاً باستخدام قواعد الأولوية هذه . وإذا تداخلت الأقواس سيبدأ التقويم بالقوسين الداخليين باتجاه الخارج.

يبين المثال التالي عبارة تضم عدة معاملات رياضية. وسنبسط العبارة في كل خطوة بواسطة تطبيق العملية التي لها الأولوية العليا:

y =	5	
x =	$24 - \neg y \wedge 5 / 4 \wedge 2 \% 12 * 6$	( تناقص قبل )
a.	$24 - 4 \wedge 5 / 4 \wedge 2 \% 12 * 6$	(الأسس)
b.	$24 - 1024 / 4 \wedge 2 \% 12 * 6$	(الأسس)
c.	$24 - 1024 / 16 \% 12 * 6$	(القسمة)
d.	$24 - 64 \% 12 * 6$	(الباقى)
e.	$24 - 4 * 6$	(الضرب)
f.	$24 - 24$	(الطرح)
g.	0	

سنستخدم فيما يلي المثال ذاته مع إضافة بعض الأقواس لتغيير ترتيب الأولوية:

y =	5	
x =	$(24 - \neg y) \wedge (5 / 4 \wedge (2 \% 12)) * 6$	(prefix decrement)
a.	$(24 - 4) \wedge (5 / 4 \wedge (2 \% 12)) * 6$	(innermost parentheses modulus)
b.	$(24 - 4) \wedge (5 / 4 \wedge 2) * 6$	(parentheses, exponentiation)
c.	$(24 - 4) \wedge (5 / 16) * 6$	(division)
d.	$(24 - 4) \wedge .31 * 6$	(parentheses, subtraction)
e.	$20 \wedge .31 * 6$	(exponentiation)
f.	$2.55 * 6$	(multiplication)
g.	15.30	(multiplication)

## المعاملات العلائقية Relational Operators

إن المعاملات العلائقية ضرورية في أي لغة برمجة مثل المعاملات الرياضية . فهي تستخدم لمقارنة سلسلتين من البيانات وإعادة قيمة منطقية "صائب منطقي" (T.) أو "خاطئ منطقي" (F.) اعتماداً على نتائج تلك المقارنة.

يمكننا في برنامج كليبر 5.2 إجراء المقارنات التالية:

العامل	الاختبار	انواع البيانات الممكنة
<> أو != أو #	لايساوي	C, D, L, N, M, NIL
<	أقل من	C, D, L, N, M
<=	اقل من أو يساوي	C, D, L, N, M
=	يساوي	C, D, L, N, M, NIL
==	يساوي تماماً	C, D, L, N, M, A, O, NIL
>	أكبر من	C, D, L, N, M
>=	أكبر من أو يساوي	C, D, L, N, M
\$	سلسلة فرعية	C, M
C = حرفي	L = منطقي	
D = تاريخ	N = رقمي	
M = مذكرة	A = منظومة	
O = هدف	NIL = صفريّة	

## المساواة Equivalence ( "==" و "=" )

معظم المعاملات المذكورة أعلاه واضحة ، إلا أنه مازال الكثير يخطئ في التمييز بين "==" (يساوي) و "==" (يساوي تماماً) . يختلف هذان المعاملان في حال مقارنة سلاسل من الرموز . فإن فحص المساواة "==" يتبع القواعد الثلاث التالية استناداً إلى طول السلسلتين المقارنتين:

? <string1> = <string2>

إذا كانت السلسلة الثانية < string2 > بلا قيمة null أو فارغة ( " ) تكون القيمة المرجعة "صائب" ( . T . ) .

وإذا كانت السلسلة الثانية < string2 > أطول من الأولى < string1 > تكون القيمة المعادة "خاطئء" ( . F . ) .

وإلا سيقارن كل حرف من السلسلة الثانية < string2 > بنظيره في السلسلة الأولى <string1> . وإذا تطابقت كافة حروف السلسلة الثانية < string2 > مع السلسلة الأولى < string1 > فستكون القيمة المرجعة "صائب" ( . T . ) وإلا ستكون القيمة المرجعة "خاطئء" ( . F . ) .

وعلى عكس ذلك يعيد فحص المساواة التامة ( " == " ) قيمة "صائب" ( . T . ) إذا تطابقت السلسلتان تماماً.

ملاحظة

ملاحظة : إذا ضبطنا " SET EXEACT ON " بحالة تشغيل ، فستنفذ كافة تدقيقات المساواة ( " == " ) بطريقة المساواة التامة ( " == " ) ذاتها . ويجب تذكر ذلك للحالات التي يجب فيها تدقيق المساواة التامة.

والاختلاف الآخر بين " = " و " == " هو أننا يمكننا استخدام معامل " = " لفحص عبارتي مصفوفة للتأكد من تساويهما . فإن أشارت كلتا العبارتين إلى المصفوفة ذاتها سيعيد فحص المساواة التامة " == " قيمة " حقيقي " ( . T . ) .

## أولوية المعاملات العلائقية

على عكس المعاملات الرياضية ، تأخذ جميع المعاملات العلائقية مستوى الأولوية ذاته ، وبالتالي يتم التقييم بالترتيب من اليسار إلى اليمين . ويمكننا مع ذلك تجاوز هذا الترتيب باستخدام القوسين .

وفي برامج أكبر ، يتم تقييم المعاملات العلائقية بعد كافة المعاملات الرياضية .

```
? 5 > 4
? 5 >= 4 // .T.
? 5 <> 4 // .T.
? 5 <> DATE() // run-time error-type mismatch
? .T. <> .T. // .F.
? date() < date() + 10 // .T.
? x < NIL // run-time error-cannot apply to NIL
? "hi " = "hi there" // .T. -left expression longer than right
? "hi there " == "hi" // .F.
? "A" < "a" // .T.
? "the" $ "hi there" // .T.
? "THE" $ "hi there" // .F. -search is case-sensitive
```

## المعاملات المنطقية Logical Operators

تتاز المعاملات المنطقية عن المعاملات الرياضية والعلائقية بأنها تمكننا من " ضم " مجموعات من عبارات منطقية مع بعضها . فيمكننا بالتالي تقييم مجموعات العبارات هذه بقيمة " حقيقي " ( . . T . ) أو " غير حقيقي " ( .F. ) واحدة . ويمكن الاستفادة منها بوجه



خاص عند الحاجة للإيفاء بشروط متعددة . وأكثر مايفيد استخدام المعامل المنطقي NOT ( ! أو .NOT.) لتغيير قيمة متغير منطقي.

العامل	الاختبار	أنواع البيانات الممكنة
.AND.	المنطق "و" AND	المنطقية
.OR.	المنطق "أو" OR	المنطقية
.NOT. !	المنطق "لا" NOT (نفي)	المنطقية

يمكن استخدام هذه المعاملات مع حقول ومتغيرات منطقية. ولكننا سنستخدمها أكثر مع العبارات التي تعيد بعد التقييم قيمة منطقية ، كما في المثال التالي:

```
customer->balance <= 250.00
EOF( )
RECNO( ) < 2000
```

### الاختصار المنطقي Logical Short-Circuiting

عندما يقيم كليبر 5.2 معاملي (.AND.) ، وتكون قيمة المعامل الأول "غير حقيقي" (.F.) فسيعيد البرنامج مباشرة قيمة "غير حقيقي" (.F.) دون الحاجة لتقييم المعامل الثاني وهذا يمكننا من كتابة شيفرة كالتالية:

```
IF VALTYPE(NAME) == "C" .AND. NAME == "the value"
    // perform action
ENDIF
```

إن لم يكن المتغير NAME من النوع الحرفي ، ستعيد الوظيفة VALTYPE( ) قيمة غير "C". ولن يقيم برنامج كليبر 5.2 المعامل الثاني.

وعلى غرار ذلك ، إذا استخدمنا معامِل (OR.) وكانت قيمة المعامِل الأول "حقيقي" (T.) فسيُعَد برنامج كليبر 5.2 قيمة "حقيقي" (T.) دون الحاجة لتقييم أي معامِل.

## أولوية المعامِل المنطقية

تقيم المعامِل المنطقية بالترتيب التالي:

١ - NOT .

٢ - AND .

٣ - OR .

عندما تستخدم أكثر من معامِل منطقي واحد في التعبير ، يتم تقييم كل "تعبير ثانوي" في مستوى الأولوية قبل التعبيرات الثانوية في المستوى التالي . وتنفذ جميع العمليات بالترتيب من اليسار إلى اليمين.

يمكننا أيضاً استخدام الأقواس لتجاوز هذا الترتيب الافتراضي . كما يتم تقييم أي "تعبير ثانوي" ضمن قوسين أولاً باستخدام قواعد الأولوية هذه وإذا تداخلت الأقواس وسيبدأ التقييم بالقوسين الداخليين باتجاه الخارج.

يتم تقييم المعامِل المنطقية بعد كافة المعامِل الرياضية والعلائقية.

## معامِل التعيين Assignment Operators

لا تكون لأي من المعامِل المذكورة أعلاه قيمة إن لم نكن نستطيع تعيين نتائجها إلى متغير أو حقل قاعدة بيانات. ولهذا الغرض يتضمن كليبر معامِل التعيين.

العامل	الوظيفة	أنواع البيانات الممكنة
=	تعيين بسيط	الكل
:=	التعيين المباشر	الكل
<op>=	التعيين المركب	C, D, N, M

### التعيين المباشر In-line Assignment

يستخدم معامل التعيين المباشر ( := ) لتعيين قيم للمتغيرات والحقول من أي نوع كانت. ويتم تقييم التعبيرات التي على يمين إشارة المساواة ، ثم تعين قيمة هذا التعبير إلى المتغير الذي على يسار النقطتين. ويمتاز التعيين المباشر عن التعيين البسيط ( = ) بأنه يمكن استخدامه حيثما يمكن كتابة تعبير أو ثابت ، بما في ذلك عبارات إعلان المتغير.

عندما نعلن المتغيرات ، يمكننا استخدام معامل التعيين المباشر لتجهيزها كما يلي:

```
static counter := 1
local oldrow := row( )
```

ويمكن استخدام التعيين المباشر لتعيين القيمة ذاتها لمتغيرات أو حقول متعددة:

```
Customer->fname := mname := "Justin"
```

وهذا مشابه لـ: [ STORE < exp > to < ver > , < varl > ] باختلاف وحيد هو أنه يمكننا تعيين قيم لحقول قاعدة البيانات إضافة إلى المتغيرات. لاحظ أننا يجب أن نستهل التعيين المباشر لحقل قاعدة البيانات بالنسخة المكافئة الملائمة.

اعتبارات تتعلق بالشبكة

إن استخدام التعيين المباشر للحقول لا يعني تجاوز ضرورة إقفال السجل.

## التعيين البسيط مقابل التعيين المباشر

إذا كنا نريد فقط تعيين قيمة إلى متغير ما ، فيمكننا استخدام أحد معاملي التعيين البسيط والمباشر لأنهما يؤديان الغرض ذاته. ولكن يجب علينا ألا نستخدم معاملي التعيين المباشر حيث يمكن أن توجد عبارة منطقية.

والأمثلة على هذه الحالة هي عبارات " CASE " و " IF " التي يجب أن تستخدم مع قيم منطقية. وسيؤدي استخدام " = " بدلاً من " = " إلى مشكلة أثناء التشغيل:

```
#include "inkey.ch"
key = inkey(0)
DO CASE
    CASE key := K_ENTER          // wrong
    *
ENDCASE
IF mvar := 25                    // wrong
    *
ENDIF
```

وبعبارة أخرى ، يعيد معاملي المساواة ( " = " ) قيمة منطقية تناسب عبارتي " CASE " و " IF " تماماً. ولكن معاملي التعيين المباشر يعيد القيمة المعينة مهما كانت. فمثلاً: تعيد عبارة `KEY := K_ENTER` القيمة (13) وهي غير منطقية ، مما سيؤدي مباشرة إلى حدوث خطأ أثناء التشغيل ( لاحظ أن `K_ENTER` ثابت يمثل قيمة مفتاح Enter في الوظيفة ( INKEY ) ).

## معاملات التعيين المركب Compound Assignment Operators

تنفذ هذه المعاملات العمليات الرياضية ثم تعين قيمة إلى المتغير أو الحقل. وصيغتها " $<op>=$ " حيث  $<op>$  هي العملية التي ستنفذ.

العملية	تكافئ لـ:	نتيجة العملية
$x += y$	$x := x + y$	الاتحاد أو الجمع
$x -= y$	$x := x - y$	الاتحاد أو الطرح
$x *= y$	$x := x * y$	الضرب
$x /= y$	$x := x / y$	القسمة
$x \% = y$	$x := x \% y$	باقي القسمة
$x ** = y$	$x := x ** y$	الأسس

وفيما يلي عدة أمثلة لمعاملات التعيين المركب:

```
string1 = "CA-Clipper "
? string1 += "5.2"           // "CA-Clipper 5.2"
xx := yy := 10
? xx += yy                   // 20
xx := yy := 10
? xx *= yy                   // 100
xx := yy := 10
? xx /= yy                   // 1
mdata := CTOD("10/01/90")
? mdata += 14                // displays 10/15/90
```

## أولوية معاملات التعيين

مثل المعاملات العلائقية ، تأخذ جميع معاملات التعيين مستوى الأولوية ذاته. لكنها تختلف عن جميع المعاملات الأخرى في أنه يتم تقييمها من اليمين إلى اليسار.



أما معاملات التعيين المركب فتختلف من ناحية أنها تجمع بين المعاملات الرياضية ومعاملات التعيين . وبالتالي ، فإن الجزء الرياضي من معامل التعيين المركب ستكون له أولوية المعاملات الرياضية الأخرى ذاتها . أما جزء معامل التعيين فستكون له أولوية معاملات التعيين الأخرى ذاتها.

وفي برامج أكبر ، يتم تقييم معاملات التعيين فقط بعد تقييم كافة المعاملات الرياضية والعلائقية والمنطقية.

## معاملات الأغراض الخاصة Special Purpose Operators

يتضمن برنامج كليبر 5.2 أيضاً مجموعة من معاملات الأغراض الخاصة التي لا تصنف تحت أي من الفئات الواردة أعلاه.

العامل	التوضيح
&	عامل ماكرو Macro Operator
->	عامل استعارة Alias Operator
@	التمرير بالاسناد Pass by reference
{ }	منظومة حرفية و محدد كتلة شيفرة.
[ ]	عنصر منظومة
( )	وظيفة أو تجمع

### مُعامل ماكرو Macro Operator

عندما يأتي رمز ( "&" ) قبل متغير حرفي أو عبارة بين قوسين ، فإنه يستدعي مجمع ماكرو الذي يجمع العبارات أثناء التشغيل.

وإذا استخدم رمز ( "&" ) باسم متغير ، فسيعيد مجمّع ماكرو قيمة ذاك المتغير. ويمكن أن يشمل ذلك أية تعبيرات صحيحة في برنامج كليبر ، بما في ذلك استدعاء الوظيفة ، لكنه لا يشمل الأوامر:

```
lname := "Jones"
x := "lname"
? &x      // Jones
x := "Year(ctod ( '01/01/91' ) )"
? &x      // 1991
x := "set decimals to 2"
? &x      // crash!
```

### ملاحظة

إذا كنا نرغب بتضمين استدعاء الوظائف في سلسلة حرفية بهذه الطريقة ثم نجعل مجمّع ماكرو يجمعها أثناء التشغيل ، يجب أن نتأكد من ربط هذه الوظائف في برنامجنا التطبيقي. وإذا كانت تلك الوظائف محتواة في ملف مكتبة ( LIB ) فنحتاج إلى إعلانها "خارجية" EXTERNAL في برنامج كليبر 5.2 ، أو "طلبها" REQUEST في برنامج كليبر 5.2 أو أي إصدار أحدث منه. وإلا فقد يحدث خطأ وتظهر رسالة تقول "الجزء الخارجي مفقود" "missing external" أثناء التشغيل عندما يبحث مجمّع ماكرو عن وظيفة غير موجودة.

إذا استخدم رمز ( "&" ) مع عبارة بين قوسين ، فسيتم تقييم العبارة أولاً ثم يعمل مجمّع ماكرو على النتيجة . نستخدم في المثال التالي وظيفة ( INDEXKEY ) في برنامج كليبر . تعيد هذه الوظيفة العبارة الرئيسية لملف فهرس معين. وبتمرير الصفر zero إلى هذا الوظيفة ( INDEXKEY() ) سيوجه لاستخدام فهرس التحكم الحالي.

```
use customer
index on lname to customer
? indexkey(0)    // lname
? &(indexkey(0)) // "Jones"
```

يمكننا تبديل النص ضمن السلاسل الحرفية باستخدام معامل ماكرو.

```
fname := "Brutus"  
? "Et Tu, &fname" // Et Tu, Brutus
```

ولكن لاجدوى من ذلك. فبدلاً من استخدام مجمع ماكرو ، من الأفضل أن نسلسل السلاسل الحرفية:

```
fname := "Brutus"  
? "Kt Tu, " + fname
```

تحذير

لا يمكن توسيع المتغيرين " ثابت STATIC " و " محلي LOCAL " بواسطة معامـل ماكرو.

ويبين المثال التالي ذلك:

```
local lname := "Jones"  
x := "lname"  
? &x // run-time error
```

لاحد لميزات ماكرو ، لكن له بعض السيئات وأعظمها أنه بطيء.

## تجميع كتل الشيفرة أثناء التشغيل

يمكن أيضاً استخدام رمز ( "&" ) لتجميع كتل الشيفرة code blocks أثناء التشغيل. ويمكن بعد ذلك حفظ كتلة الشيفرة الناتجة في متغير لتقييمها لاحقاً.

```
myblock := &("{ | a | qout(a + 1) }")  
eval(myblock, 5) // 6
```

وهكذا يمكننا تجميع كتل الشيفرة لمرة واحدة والإشارة إليها عدة مرات في برنامجنا. وبمقارنة ذلك مع ماكرو ، نجد أنه يجب إعادة تجميع ماكرو كلما أشرنا إليه.

## معامل النسخة المكافئة Alias Operator

إذا سبقنا معامل النسخة المكافئة ( ">" ) بنسخة مكافئة صحيحة وكتبنا بعده عبارة ما ، فإنه يمكننا من الإشارة إلى حقل ، أو تقييم تعبير في منطقة العمل المحددة من قبل النسخة المكافئة.

وإن إشارة النسخة المكافئة إلى منطقة عمل غير مختارة ، سيختار معامل النسخة المكافئة منطقة العمل المطلوبة تلقائياً ، وينفذ العملية ، ويختار منطقة العمل السابقة ثانياً. وبعبارة أخرى: بدلاً من كتابة العبارات الثلاث التالية:

```
select invoice
seek "12345"
select customer
```

يمكننا كتابة عبارة واحدة:

```
invoice->( dbseek("12345") )
```

إذا استخدمنا عبارة بهذه الطريقة ، يجب أن نضعها ضمن قوسين:

```
invoice->(eof( ) )      // proper
invoice->eof( )         // will not compile
```

بحثنا سابقاً الأوامر العديدة التالية المتعلقة بقاعدة البيانات والوظائف المرتبطة بها:

الوظيفة	الأمر المطابق له
dbAppend( )	APPEND BLANK
dbClearFilter( )	SET FILTER TO
dbClearIndex( )	SET INDEX TO
dbClearRel( )	SET RELATION TO
dbCloseArea( )	USE
dbCommitAll( )	COMMIT
dbCreateIndex( )	INDEX ON ... TO ...
dbDelete( )	DELETE
dbGoBottom( )	GO BOTTOM
dbGoto(<n>)	GOTO <n>
dbGoTop( )	GO TOP
dbUseArea( . . . )	USE <n> . . .

الجدول مستمر من الصفحة السابقة....

الوظيفة	الأمر المطابق له
dbRecall( )	RECALL
dbReindex( )	REINDEX
dbSeek(<exp>)	SEEK <exp>
dbSelectArea(<n>)	SELECT <n>
dbSetindex( [<n>] )	SET INDEX TO [<n>]
dbSetFilter( )	SET FILTER TO . . .
dbSetOrder(<n>)	SET ORDER TO <n>
dbSetRelation( )	SET RELATION TO . . .
dbSkip([<n>])	SKIP [<n>]
dbUnlock( )	UNLOCK
dbUnlockAll( )	UNLOCK ALL
dbPack( )	PACK ( use with caution ! )
dbZap( )	ZAP ( use with caution ! )

إذا استخدمنا هذه الوظائف الجديدة مع معاملات النسخ المكافئة يمكننا أن نجعل برنامجنا أقرب إلى الكمال وأوضح لأنه يتطلب عبارات اختيار SELECT أقل عليية. فمثلاً يفتح البرنامج التالي قاعدة بيانات رئيسية وأخرى فرعية ، ويلف حول قاعدة البيانات الرئيسية ويحذف كافة السجلات الفرعية المرتبطة بكل قاعدة رئيسية.

```

use child index child new
use parent new
do while .not. eof ( )
    select child
    seek parent -> name
    if found ( )
        do while child-> name == parent-> name
            delet
            skip
        enddo
    endif
    skip
enddo

```

إن الخطأ الذي ارتكبناه هنا هو أننا لم نعد اختيار منطقة العمل الرئيسية في آخر حلقة DO WHILE الرئيسية.



باستخدام معامل النسخة المكافئة ووظائف قاعدة البيانات يمكننا أن نجعل برنامجنا خالياً تقريباً من الأخطاء:

```
use child index child new
use parent new
do while .not. parent->(eof( ))
  child->(dbseek (parent->name) )
  if child->(found( ))
    do while child-> name == parent-> name
      child->(dbdelete( ))
      child->(dbskip( ))
    enddo
  endif
  parent->(dbskip( ))
enddo
```

إن ذلك أفضل من الحاجة لتذكر إعادة اختيار منطقة العمل ، كما أوضحنا سابقاً، ويمكننا مباشرة تحديد منطقة العمل الملائمة لعملية معينة.

## تمرير المتغيرات بالإشارة Pass by Reference

تمرر المتغيرات عادة إلى الوظائف بواسطة قيمة. وهذا يعني أن الوظيفة تنسخ المتغير. وإن أية تعديلات تجرى على المتغير الذي في الوظيفة تؤثر على النسخة فقط.

```
mver = 5
myfunc(mvar)
? mvar          // 5

function myfunc(x)
? ++x           // 6
return nil
```

لاحظ أن قيمة MVAR لا تتغير في الوظيفة بمستوى أعلى.

أما عندما نستعمل المتغير برموز "@" فإننا نمرره بالإشارة. وبدلاً من تمرير قيمة المتغير، نمرر عنوان الذاكرة الفعلي التي تم تخزين المتغير فيها. وبذلك ستؤدي أية تغيرات في المتغير من وظيفة بمستوى أخفض إلى تغيير المتغير مباشرة. وسنعيد كتابة المثال المبين أعلاه كما يلي:

```
mvar = 5
myfunc (@mvar)
? mvar           // 6

function myfunc(x)
? ++x             // 6
return nil
```

علماً بأننا نمرر المتغيرات بواسطة قيمة، فهناك بعض الحالات التي نحتاج لتمثيل المتغيرات بالإشارة بحيث يمكن تغييرها بواسطة الوظيفة التي نمررها إليها. وتمرر حقول قاعدة البيانات دائماً بواسطة قيمة لأنها ليس لها بذاتها عنوان ذاكرة.

تتعامل المصفوفات بطرق تختلف نوعاً ما عن المتغيرات، إذ أننا دائماً نمرر المصفوفات بأكملها بالإشارة، وهذا يعني أنها يمكن أن تغير في وظيفة بمستوى أخفض. وإذا استطعنا تمرير مصفوفة بأكملها بواسطة قيمة، فسيؤدي ذلك إلى أن تعمل الوظيفة نسخة محلية لكل عنصر في المصفوفة. وهذا ليس عملياً عندما تحتوي المصفوفات على مئات (أو آلاف) العناصر.

أما عناصر المصفوفات فتمرر دائماً بقيمة. وإن حاولنا كتابة رموز "@" قبلها فسيؤدي ذلك إلى خطأ في المجموع.

إن الوظيفة التي مررنا إليها العنصر تعمل نسخة من عنصر المصفوفة وتعالج هذه النسخة بدلاً من الأصل.

## القوسان الحاصران { }

يستخدم هذان القوسان للإشارة إلى بداية مصفوفة حرفية أو كتلة شيفرة. إن برنامج كليبر يميز بين المصفوفة والكتلة بوجود رمز العمود ( " | " ) الذي يستخدم لعرض قائمة متغيرات كتلة الشيفرة. وفيما يلي أمثلة على المصفوفات والكتل:

```
local array1 := { 0, .T., date( ), space(50) }
static array2 := { "Jennifer", "Greg", "Justin" }
local block1 := { | myfunc( ) }
local block2 := { | a, b | qout(a, b) , a += b }
static array3 := { 1, 2, 3, 4, space(10) } // wrong
```

لاحظ أن إعلان (ARRAY3) لن يجمع ، والسبب هو استدعاء وظيفة "مسافة" ( ) SPACE في برنامج كليبر. عندما نعلن متغيرات ومصفوفات ثابتة ، يمكننا استخدام ثوابت فقط. ولا تُقبل استدعاءات الوظائف ، باستثناء بعض الوظائف التي تكون متغيرات أثناء التجميع (مثل: ( ) CHR و ( ) ASC).

## القوسان المعقوفان [ ]

يدل هذان القوسان على أن هناك إشارة إلى أحد عناصر المصفوفة ، ويجب أن يسبقهما إشارة إلى المصفوفة أو المصفوفة الحرفية. كلا المثالين التاليين صحيح وسيعرض الاسم المطلوب ذاته:

```
local a := { "Jennifer", "Greg", "Justin" }
? a[3]
? { "Jennifer", "Greg", "Justin" } [3]
```

إذا أردنا الإشارة إلى عنصر من مصفوفة متداخل فيمكننا ذلك بطريقتين: [x,y] أو [x][y]. في المثال التالي ستعرض كلا العبارتين رقم 5: وهو العنصر الثاني في المصفوفة الفرعية الموجودة عند a[4].

```
local a := { 1, 2, 3, { 4, 5, 6 } }
```

local x := 4

local y := 2

? a[x, y]

? a[x] [y]

## القوسان ( )

ذكرنا آنفاً أنه يمكن استخدام القوسين لتجاوز قواعد الأولوية عند تقويم المعاملات.

ولكن يجب ألا ننسى أن القوسان يشيران أيضاً إلى الوظائف عندما يسبقهما اسم وظيفة

صحيح.







## التجميع Compiling

### القاعدة اللغوية Basic Syntax

لتجميع ملف (.PRG) يجب أن ننفذ الأمر CLIPPER متبوعاً باسم الملف وأية خيارات نريد استخدامها. وليس من الضروري تعيين ملحق ملف برنامج كليبر (.PRG) مع الأمر التالي (MYPROG) وسيجهز ملف الهدف MYPROG.OBJ إن لم تحدث أية أخطاء توقف التجميع.

clipper myprog

### عملية التجميع

تتكون عملية التجميع في برنامج كليبر 5.2 من خطوتين:

- قبل تنفيذ عملية التجميع لبرنامج المصدر الذي نعدّه نحن ، يقوم المعالج الأولي المدمج في برنامج كليبر 5.2 بالبحث في برنامجنا ( .PRG ) عن أية موجهات ويترجمها إلى برنامج المصدر الذي يمكن تجميعه.

- تمرر مخرجات المعالج الأولي إلى المجمع الذي يجهز برنامج الهدف (على افتراض أنه لا يواجه أخطاء في برنامجنا). ويمكننا إذا أردنا إعادة توجيه مخرجات المعالج الأولي إلى ملف فيه خيار المعالج ( /P ) بحيث يمكننا أن نرى فعلياً كيف يربك المعالج الأولي برنامج المصدر الذي نعدّه.

## خيارات المجمع

يتيح مجمع برنامج كليبر 5.2 استخدام مجموعة كبيرة من الخيارات في سطر الأوامر كما أن جميع مفاتيح المجمع لا تميز بين حالة الحرف كبير أم صغير case-insensitive.

التركيبة اللغوية	التوضيح
/a	اعلان متغيرات ذاكرة تلقائية
/b	تفلية المعلومات
/credits	شاشة الشكر والثناء
/d<id>[=<val>]	تعريف المعرف #define <id>
/ES	تهيئة المجمع لمستوى خطورة الخروج (الاصدار 5.2 فقط)
/i<path>	تضمين ممر البحث عن الملف في #include
/l	إخماد ترقيم سطور المعلومات
/m	تجميع وحدة واحدة فقط
/n	لا يتضمن اجراء البداية
/o<path>	سواقة ملف الهدف و / أو الممر
/p	توليد ملف مخرجات المعالج الأولي (.ppo)
/q	إخماد عرض أرقام السطور
/r[<lib>]	يطلب من الرابط البحث عن <lib> أو لا.
/s	فحص صحة التركيب اللغوي فقط
/t<path>	تحديد موقع الملفات المؤقتة
/u[<file>]	تعريف ملف ترويسة قياسي بديل للمعالج الأولي في <file> أو لا.
/v	يفترض أن كل المتغيرات في الملف المجمع هي M->
/w	توليد رسائل التحذيرات
/z	ايقاف المختصرات المنطقية

أهم هذه الخيارات هي ( /B ) و ( /N ) و ( /W ).

## مفتاح ( /b ) تضمين معلومات برنامج كشف الأخطاء

يحتوي كليبر 5.2 برنامجاً لكشف الأخطاء على مستوى ملف البرنامج المصدر- (source level debugger) الذي يسهل عملية كشف الأخطاء بأن يتيح لنا مشاهدة برنامج المصدر الذي نعدّه أثناء تنفيذ برنامجنا. ولكن هذا يعني أننا يجب أن نعدّ برنامجنا بشكل مختلف قليلاً إذا أردنا كشف الأخطاء فيها.

إذا أردنا كشف الأخطاء في برنامج معدّ في برنامج كليبر 5.2 ، فيجب علينا تضمين مفتاح ( /b ) الذي يضمن معلومات برنامج كشف الأخطاء في ملف الهدف. وعلنا أيضاً أن نترك أرقام الأسطر في ملف الهدف. وبعبارة أخرى ، إذا استخدمنا خيار ( /b ) يجب ألا نستخدم خيار ( /l ).

إن أردنا كشف الأخطاء في برنامجنا ، يمكننا تجميع بعض الوحدات فقط بواسطة مفتاح ( /b ). وإذا شغلنا برنامجنا بعد ذلك مع برنامج كشف الأخطاء debugger فسيتوقف هذا الأخير عند الوحدات المجمعة بواسطة مفتاح ( /b ) فقط.

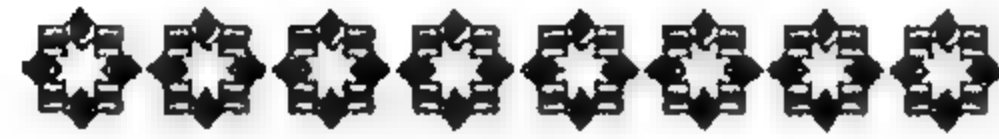
## مفتاح ( /n ) منع إجراء بداية التشغيل

يمنع هذا الخيار التعريف الآلي لإجراء معين يحمل اسم الملف ذاته (PRG). وإذا أردنا استخدام المتغيرات الساكنة للملف الواسع file-wide ، فيجب أن نستخدم خيار هذا التجميع.

## مفتاح ( /w ) إصدار رسائل تحذيرية

يأمر هذا الخيار المجتمع بإصدار رسائل تحذيرية لإشارات المتغيرات غير المعلنة أو غير المميزة ("الغامضة"). يجب استخدام هذا الخيار كلما جتمعنا باستخدام مجمع كليبر 5.2. وبذلك يحذرنا المجمع كلما نسينا إعلان متغير ما ، وهذا سيجعل برنامجنا أفضل بكثير.

الفائدة الأخرى من استخدام مفتاح ( /W ) هو أنه يمكن المجمع من التحذير عند وجود أخطاء طباعية. وهذا يعني أننا يمكننا تصحيح الأخطاء الطباعية قبل مرحلة الربط ثم التنفيذ.



## الربط Linking

لقد كتبنا برنامج المصدر وعالجناه في المجمع لتجهيز ملفات الهدف (.OBJ) وسنقوم الآن بتجميع هذه الملفات في ملف " تنفيذي " (.EXE) واحد قابل للتنفيذ. وتعرف هذه العملية بالربط. وبرنامج الربط المتوفر في برنامج كليبر ٥,٢ هو RTLINK. التركيب اللغوي الافتراضي لبرنامج RTLINK هو صيغة غير محددة كما يلي:

```
rtlink fi <object1> [, <object2>] [ li <lib1> [, <lib2> . . . ] ]
```

لا حاجة لتعيين ملحق ملف الهدف (.OBJ) أو (.LIB). بل لا حاجة لتعيين أي من المكتبات الأربعة المتوفرة مع برنامج كليبر EXTEND.LIB , CLIPPER.LIB , DBFNTX.LIB , TERMINAL.LIB لأن برنامج الربط سيجد طلبات البحث عنها مضمنة في ملفات الهدف الذي أعدناه.

فعلى سبيل المثال إذا كان لدينا ثلاثة ملفات (DATA.PRG) , (REPORTS.PRG) , (MAIN.PRG) فستكون عبارة الربط كما يلي:

```
rtlink fi main,date,reports
```

من المهم الترتيب عند تعيين ملفات (.OBJ). ويجب التأكد أن الملف الأول يحتوي وظيفة الإدخال.

## الإحلال Overlays

إن الإحلال هو طريقة تسمح بتنفيذ البرامج الكبيرة بذاكرة أقل من الذاكرة اللازمة عادة. وإن أجزاء البرنامج التي تحول عادة إلى ملف (.EXE) توجه بدلاً من ذلك إلى أقسام منفصلة. ولا تحمل هذه الأقسام المنفصلة مباشرة أثناء التشغيل. بل إنها تسحب إلى الذاكرة فقط عندما يستدعي إجراء أو وظيفة موجود ضمنها بواسطة برنامجنا. وهذا سينقص حجم التحميل المطلوب لبرنامجنا بشكل كبير. وعلى سبيل المثال ، إذا كان

حجم البرنامج ٤٠٠ ك ، يمكننا نقل ١٤٠ ك من البرنامج إلى منطقة إحلال فسيخفض حجم التحميل إلى ٢٦٠ ك.

إن برنامج RTLINK (المتوفر في برنامج كليبر 5.2) يضع تلقائياً كافة أجزاء البرنامج المعد في برنامج كليبر في إحلال ديناميكي dynamic overlay. يتكون الإحلال الديناميكي من برنامج يتم تحميله في الذاكرة (أو "مساحة الإحلال") أثناء التشغيل ، ويتم الإحلال الديناميكي عموماً بتحميل البرنامج على مستوى الإجراء أو الصفحة ، وليس على مستوى الهدف. وهذا يعني أنه ليس من اللازم الاهتمام بتركيب الإحلالات كما هي الحال في لغات البرمجة الأخرى (والإصدارات السابقة لبرنامج كليبر).  
ومهما كان حجم الشيفرة برنامج كليبر ، فإنها ستنفذ في مساحة إحلال صغيرة نسبياً مما سيوفر لبرنامجنا مساحة أكبر للعمل في الذاكرة.

## برامج ربط من شركات أخرى

هناك نقطتان تحذان من عمل ربط الإحلال الديناميكي في برنامج RTLINK:

- لا يستخدم إلا مع برامج مجمعة في برنامج كليبر. وإن وحدات Modules برامج لغة " C " و " Assembler " يجب أن توجد في المنطقة الجذرية الأساسية (مع مدير الإحلال لـ RTLINK في برنامج كليبر 5.2) أو في إحلال ساكن (static overlay).
- مع أنه يوفر الربط التزايد إلا أنه دون الحد القياسي. (يتيح الربط التزايد حلقات ربط أسرع بواسطة إدراج وحدات برامج ( OBJ . ) التي تغيرت فقط في ملف "تنفيذي" ( EXE . ).

ولهذا السبب قد نستخدم أحد برنامجي الربط اللذين سنبحثهما فيما يلي. إذ أنهما يوفران ربطاً تزايدياً سريعاً. وسيبدلان ديناميكياً معظم أهداف لغة سي ولغة Assembler إضافة إلى كافة وحدات برامج أهداف كليبر ، ولكن توجد تحذيرات عامة وهي:



- يجب أن يستخدم البرنامج واجهة نظام التوسيع Extend System interface في برنامج كليبر لتمرير المتغيرات ، ولا يعتمد على أية ميزة غير موثقة من برنامج كليبر.
- يجب ألا يعالج البرنامج أي نوع من المقاطعات interrupts.
- لا يمكن أن يكون البرنامج قادراً على التعديل الذاتي.

### برنامج الربط (Blinker)

إنه برنامج ربط لا يعتمد على اللغة ومصمم ليستخدم في البرامج التطبيقية لإصدار Summer'87 وكليبر 5.2. وهو سريع جداً ويوفر ربطاً تزايدياً سريعاً ، وتتوافق قاعدته اللغوية مع برنامج RTLINK. كما أنه يمكننا من تضمين أرقام متسلسلة ومعلومات بيئية لبرنامج كليبر مباشرة في الملف القابل للتنفيذ الخاص بنا.

يمكننا برنامج ( Blinker ) من تجهيز نسخ تجريبية لبرامجنا التي لها حد استخدام يعتمد على التاريخ أو طول زمن التنفيذ أو عدد مرات الاتصال بمدير الإحلال. كما يتضمن ميزة هامة يمكننا من رفع مستوى تتابع القواعد المنطقية لبرامجنا. إنه يساند عملية دمج جدول الرموز. وقد أضيفت في إصدار 2.0 من هذا البرنامج ميزة تبادل نظام التشغيل DOS - Swapping التي يمكننا من تنفيذ برامج كبيرة من داخل برنامج كليبر دون نفاذ الذاكرة.

### برنامج الربط ( Warplink )

إنه برنامج ربط لا يعتمد على اللغة أيضاً ، وهو متوافق مع إصدار Summer'87 وبرنامج كليبر 5.2. كما أنه يربط ويقوم بالإحلال الديناميكي لعدد من اللغات مثل

سي و Assembler و BASIC و FORTRAN و COBOL ، كما أنه سريع جداً ،  
ويساند القاعدة اللغوية RTLINK POSITIONAL.

يحتوي برنامج ( Warplink ) على تقنية موجودة في مكتبة برنامج "الذاكرة  
الذكية" SMARTMEM ، التي تتيح صورة جانبية شاملة عن الذاكرة وقدرة تجميع  
أجزائها. ويمكننا من دمج جدول الرموز. وتتوفر فيه أيضاً إمكانية محاكاة الشروط  
الفعلية بواسطة الحد من مقدار الذاكرة المتوفرة للاستخدام من قبل برنامجنا.

يستطيع برنامج ( Warplink ) تجهيز ملفات COM. مباشرة من ملفات الهدف لبرامج  
ليست مكتوبة بكليبر ، أي في الحالات التي تستخدم فيها مثلاً لغة سي أو  
Assembler ، كما أنه يساند أيضاً الذاكرة الموسعة ( EMS ) ونظام ( XMS ) أثناء  
الربط وأثناء التشغيل. وقد أضيفت إليه في الوقت الراهن الميزات التالية: التجزئة ،  
وإمكانية فصل المكتبات إلى وحدات برامج مستقلة للإحلال ، ومكتبات ربط ديناميكية  
فعلية.



## أساليب إضافة الملاحظات

يمكننا برنامج كليبر 5.2 ، مثل معظم لغات البرمجة ، من إضافة ملاحظات في برنامجنا بحرية بواسطة إضافة عبارات لا تنفذ. ولتجهيز ملاحظة من سطر واحد يمكننا أن نستعملها بـ "//" أو "\*" . وإذا أردنا إضافة ملاحظة في سطر شيفرة البرنامج ذاته يمكننا أن نستعملها بـ "//" أو "\*" أو "&&" .

```
* this is a single line comment
// so is this
? whatever && this is an in-line comment
? whatever // so is this
```

لأفرك أبدأ بين هذه الرموز ، ولنا مطلق الحرية في اختيار "/" أو "\*" أو "&&" ، ولكن يفضل استخدام "/" "\*" عند تجهيز كتلة من الملاحظات. تشير "/" إلى بداية الكتلة ، و "/" "\*" إلى نهايتها.

```
// this is the tedious way
// to comment out multiple
// lines of code
/*
    this is the new way
    to comment multiple
    lines of code. Easier
    on the eyes & fingers
*/
```

السيئة الوحيدة في استخدام كتل الملاحظات "/" "\*" هي أنها لا تتداخل. ولكن يمكننا تجاوز ذلك بواسطة استخدام المعالج الأولي في كليبر 5.2.

إذا كان لدينا برنامج كبير نريد إيقافه مؤقتاً ، ويحتوي هذا البرنامج على كتلة أو أكثر من الملاحظات ، فيجب أن نضع موجهاً من نوع ( #ifdef ) قبل شيفرة البرنامج باستخدام اسم ثابت وهمي ، ثم نضع موجه ( #endif ) مباشرة بعد الشيفرة.

```
#ifdef BLAHBLAHBLAH
// statements
/*
comment
*/
// more statements
#endif
```

هذه الطريقة هي أسرع بكثير من استخدام ملاحظات من سطر واحد في العبارات التي نريد التأثير عليها جميعها.

## تراكيب التحكم Control Structures

توجد في برنامج كليبر 5.2 خمسة تراكيب أساسية للتحكم. ويوجد معظمها في لغات البرمجة الأخرى أيضاً. ويمكن أن يتداخل أي من هذه التراكيب ضمن ذاته أو ضمن التراكيب الأخرى شريطة إغلاق كل تركيب بشكل صحيح.

### تركيب IF...[ ELSE ]..ENDIF

يتيح هذا التركيب تنفيذ واحد من عدة كتل منطقية اعتماداً على شرط منطقي واحد أو أكثر. إذا كانت نتيجة تقييم الشرط الذي يلي العبارة الشرطية (IF) "حقيقي" (T.)، فستنفذ العبارات التي تليه مباشرة حتى تظهر العبارة (ELSE) أو (ELSEIF) أو (ENDIF). أما إن كانت نتيجة تقييم العبارة الشرطية (IF) "غير حقيقي" (F.)، وهناك عبارة (ELSEIF) فسيتم تقييم ذلك الشرط وسيكرر الإجراء حتى تصبح عبارة (ELSEIF) "حقيقي" أو لا يوجد عبارات (ELSEIF) أخرى.

وإن لم تكن هناك عبارة (ELSEIF) فسيمر التحكم إلى عبارة تلي عبارة (ELSE) (إذا وجدت) أو إلى عبارة (ENDIF).

إن كانت نتيجة تقييم الشرط الأول <condition1>، في المثال التالي، "حقيقي" (T.) فستنفذ العبارتان (١) و (٢) ثم يمر التحكم إلى العبارة (٧). وأما إن كانت النتيجة "غير حقيقي" (F.) فسيتم تقييم الشرط الثاني <condition2>. فإذا كانت النتيجة "حقيقي" فستنفذ العبارتان (٣) و (٤) ثم يمر التحكم إلى العبارة (٧)، وإلا فستنفذ العبارتان (٥) و (٦).

```
if <condition1>
    statement1
    statement2
elseif <condition2>
    statement3
```

```

statement4
else
    statement5
    statement6
endif
statement7

```

## تركيب DO WHILE..ENDDO

يؤدي هذا التركيب إلى تنفيذ عبارة أو أكثر في حلقة طالما كانت نتيجة تقييم الشرط المنطقي "حقيقي" (.T.). وعند الوصول إلى عبارة (ENDDO) يعود التحكم إلى عبارة (DO WHILE) التي تعيد تقييم الشرط. وفي المثال التالي ، سيعرض حقلاً "الاسم الأول" و "اسم العائلة" لكل سجل في قاعدة البيانات "العميل" CUSTOMER.

```

dbgotop( )
do while .not. eof( )
    ? customer->firstname
    ? customer->lastname
    dbskip( )
enddo

```

نفيدنا القاعدة اللغوية لـ: DO WHILE عندما نريد تنفيذ حلقة ولانعرف متى ستنتهي الحلقة.

يمكننا أيضاً استخدام أمر "الخروج" EXIT أو أمر "حلقة" LOOP ضمن حلقة (DO WHILE..ENDDO) . يؤدي أمر "الخروج" إلى مرور التحكم إلى العبارة التي تلي ENDDO. أما الأمر LOOP فيؤدي إلى عودة التحكم إلى عبارة DO WHILE. ولكن يفضل عدم استخدام أمر "الخروج" EXIT لأنه قد يجعل عملية متابعة برنامجنا أصعب. وبدلاً من استخدامه ، يجب علينا كتابة شرط الخروج في عبارة DO WHILE كما في المثال التالي:

```

// bad
do while .t.
    // statements
if condition

```



```
    exit
  endif
enddo

// good
do while .not. condition
  // statements
enddo
```

## تركيب DO CASE..ENDCASE

يشبه هذا التركيب تركيب (IF..ELSEIF..ENDIF) تماماً. فهو يتيح تنفيذ واحد من عدة كتل اعتماداً على شرط منطقي واحد أو أكثر. يتم تقييم كل شرط "حالة" CASE ابتداءً بالأول. فإذا كانت نتيجة تقييم الأول "حقيقي" (.T.) فستنفذ العبارات التي تليه مباشرة حتى تظهر عبارة (CASE) أو (OTHERWISE) أو (ENDCASE). وإن كانت النتيجة "غير حقيقي" (.F.) فسيتم تقييم شرط الحالة التالي (إن وجد) ، أو يمر التحكم إلى العبارة التي تلي عبارة (OTHERWISE).

إذا كانت نتيجة تقييم الشرط الأول <condition1> ، في المثال التالي ، "حقيقي" فستنفذ العبارتان (١) و (٢) ثم يمر التحكم إلى عبارة (٧) . وأما إن كانت النتيجة "غير حقيقي" (.F.) فسيتم تقييم الشرط الثاني <condition2> . فإن كانت نتيجة "حقيقي" فستنفذ العبارتان (٣) و (٤) ثم يمر التحكم إلى عبارة (٧) ، وإلا فستنفذ العبارتان (٥) و (٦).

```
do case
  case <condition1>
    statement1
    statement2
  case <condition2>
    statement3
    statement4
  otherwise
    statement5
    statement6
endcase
statement 7
```

## تركيب FOR..NEXT

يشبه هذا التركيب تركيب DO WHILE من حيث أنه يدور لينفذ كتلة عبارات بشكل متكرر. ولكنه يختلف عنه في أنه يجب أن يستخدم عندما نعرف سلفاً عدد المرات التي ستنفذ فيها الحلقة.

والقاعدة اللغوية لتركيب FOR.. NEXT هي:

```
FOR <counter> := <start> TO <stop> [STEP <increment>]
  // statements
NEXT
```

حيث <start> هي القيمة الأولية التي ستعين إلى العداد <counter>. وبشكل افتراضي تزيد قيمة "العداد" واحد كلما نفذت الحلقة ، حتى ولو عينا مقدار زيادة مختلف بواسطة STEP الاختيارية. يمكن أن يكون مقدار الزيادة < increment > رقماً سلبياً، وبهذه الحالة يجب أن تكون قيمة <start> أكبر من <stop>. عندما يتجاوز "العداد" <counter> القيمة المحددة بواسطة <stop> فستنتهي الحلقة.

ستؤدي العبارات التالية إلى إضافة ١٠٠٠ سجل فارغ إلى قاعدة البيانات في منطقة العمل الحالي ، وسيعرض الأرقام (١) إلى (١٠٠٠) على الشاشة:

```
// good
for x := 1 to 1000
  append blank
  ? x
next
```

كما هي الحال مع تركيب DO WHILE ، يمكننا استخدام أمر "الخروج" EXIT أو أمر حلقة LOOP ضمن حلقة FOR..NEXT ، ولكن لايفضل استخدام هذه الطريقة. يؤدي أمر "الخروج" EXIT إلى مرور التحكم إلى العبارة التي تلي عبارة NEXT. ويؤدي أمر "حلقة LOOP" إلى عودة التحكم إلى عبارة FOR.

تحذير

إذا عينا بدلاً ثابتاً كحد أعلى للحلقة ، فسيقوم برنامج كليبر 5.2 هذه العبارة كلما عاد النظام إلى أول حلقة FOR..NEXT.

ولذلك ، ولتحسين الأداء ، يفضل تقييم العبارات قبل عبارة FOR كما يلي:

```
// good  
n := lastrec( )  
for x := 1 to n
```

```
// bad  
for x := 1 to lastrec( )
```

## تركيب BEGIN SEQUENCE..END SEQUENCE

يمكن أن يستخدم هذا التركيب للحد من مشاكل البرنامج المتوقعة. إذا صدرت عبارة الإيقاف BREAK في أي مكان ضمن هذا التركيب فإنها تجبر البرنامج على الانتقال إلى العبارة التي تلي عبارة نهاية التسلسل END SEQUENCE مباشرة. لاحظ أنه ليس من الضروري إصدار أمر الإيقاف BREAK ضمن الوظيفة ذاتها مثل تركيب BEGIN SEQUENCE .. END SEQUENCE. بل يمكن إصداره من أي برنامج يستدعى من داخل العبارة BEGIN SEQUENCE..END SEQUENCE ، وبالتالي يكون برنامجنا أفضل وأوضح.

يمكننا تحسين هذا التركيب بإضافة عبارة الإصلاح RECOVER. فإن وجدت هذه العبارة وتوقف البرنامج بالأمر BREAK في نقطة ما داخل BEGIN SEQUENCE .. END SEQUENCE ، فسينتقل مسار البرنامج إلى العبارة التي تلي عبارة الإصلاح RECOVER مباشرة. وإن لم يصدر أمر الإيقاف Break ، فسينتقل مسار البرنامج من BEGIN SEQUENCE إلى عبارة الإصلاح RECOVER ، ثم يقفز

مباشرة إلى العبارة التي تلي END SEQUENCE (وبالتالي فقد تجاوز أي برنامج إصلاح).

إن تركيب BEGIN SEQUENCE .. END SEQUENCE مشابه لتركيب عبارة IF..ELSE..ENDIF ، والفارق الوحيد هو أننا في التركيب الثاني نأخذ أحد الفروع (IF) أو الآخر (ELSE) ، بينما في تركيب BEGIN SEQUENCE يمكننا تنفيذ جزء من الفرع الأول ثم ننفذ الفرع الثاني بأكمله.

لا يتسع المجال هنا لبحث معالجة الأخطاء بواسطة هدف الخطأ Error object ، لكننا سنبين في المثال التالي كيفية استخدام وظيفة "كتلة الخطأ" Errorblock لإرسال معالج خطأ عادي يصدر أمر "إيقاف" Break في حال وجود خطأ أثناء التشغيل. فإذا ظهرت مشكلة أثناء محاولة فتح قاعدة البيانات ، سيصدر خطأ أثناء التشغيل ويؤدي إلى تقييم كتلة الخطأ Errorblock. وبالتالي سيصدر أمر إيقاف BREAK سيؤدي بدوره إلى تمرير التحكم إلى العبارة التي تلي عبارة الإصلاح Recover مباشرة.

```
errorblock( { | e | break(NIL) } )
begin sequence
  // attempt to open a database
  // operate on the file
recover
  err_msg( "could not process file" )
end sequence
```



## تحديد مجال المتغيرات

عند استخدام برنامج كليبر 5.2 سنحتاج بالتأكيد إلى استخدام المتغيرات التي تحتوي على المعلومات (أي: نسخ مؤقتة من محتويات حقول قاعدة البيانات ، عدادات الحلقة معلومات البيئة للاستعادة ... الخ). عند استخدام المتغير يفضل أن تحدد مجاله. فالمجال يؤثر على مدة استخدام المتغير وعلى ظهوره ضمن البرنامج.

هناك أربعة أنواع من مجالات المتغيرات في كليبر 5.2 : "خاص PRIVATE" و "عام PUBLIC" و "محلي LOCAL" و "ساكن STATIC". ويعتبر الأولان "ديناميكيان" dynamic ، بينما يعرف "المحلي local" و "الساكن static" بألهما "تركيبيان".

الفارق الرئيس بين الديناميكي والتركيبي هو استخدام جدول الرموز في برنامج كليبر. فكلما أشار برنامجنا إلى متغير ديناميكي ، يجب أن يتبع برنامج كليبر عملية مكونة من خطوتين ليستنتج قيمته:

- أولاً : يجب أن يبحث عن اسم المتغير في جدول الرموز. وإن جدول الرموز يجهز عندما نجمع برنامجنا ، ويحتوي بشكل مبدئي أسماء كافة المتغيرات الدينامكية ، إضافة إلى أسماء أية وظائف أشرنا إليها في برنامجنا. ويحفظ جدول الرموز أثناء التشغيل أيضاً بعناوين الذاكرة التي تخزن فيها قيم كل متغير.
- وبمجرد أن يجد برنامج كليبر المتغير في جدول الرموز ، يمكنه تحديد عنوان الذاكرة المخزنة فيه القيمة. ثم ينظر إلى عنوان الذاكرة هذا ليحدد قيمة المتغير.

إن الخطوة الإضافية للنظر في القيمة تعني أن يكون الأداء أبطأ باستخدام المتغيرات الدينامكية. وعلاوة على ذلك ، سيتطلب كل متغير "عام PUBLIC" أو "خاص PRIVATE" ما يقارب ١٦ بايت في جدول الرموز. وإن برنامج كليبر يستخدم مئات بل آلاف المتغيرات. وعلى سبيل المثال ، إذا استخدمنا ٥٠٠ متغير في البرنامج وكانت



كلها ديناميكية ( أي "خاصة private" أو "عامة public" ) فإن ذلك سيزيد من حجم جدول الرموز بمقدار ٨ ك.

وبما أن حجم جدول الرموز يؤثر مباشرة في حجم الحمل فإننا سنحتاج إلى ٨ ك إضافية لتحميل البرنامج. وهذا قد يؤثر أحياناً في عمل الشبكة. الفائدة الوحيدة من المتغيرات الديناميكية هي أنها يمكن أن تحل محل الماكرو لأن معامل هذا الأخير يتطلب وجود بند في جدول الرموز. وبما أنه توجد في برنامج كليبر 5.2 العديد من البدائل لماكرو فيفضل عدم استخدام المتغيرين "خاص private" و "عام public".

#### ملاحظة

لاحظ أن الحد الأعلى المسموح به لاستخدام المتغيرات الديناميكية في برنامج كليبر ٥,٢ هو (٢,٠٤٨) في أي وقت كان. بينما يمكن أن نستخدم المتغيرين التركيبين ( المحلي Local والساكن Static ) بلا حدود.

## إعلان المتغير الخاص Private

إن المتغيرات الخاصة مرئية ضمن الوظيفة التي أعلنت فيها. كما أنها مرئية ضمن أية وظائف تستدعي من تلك الوظيفة ولكن ليست من الوظائف ذات المستويات الأعلى.

إذا أعلننا متغيراً "خاصاً PRIVATE" دون تأسيسه بقيمة ابتدائية ، فسيعين للمتغير قيمة البدء صفر NIL. وإذا أعلننا مصفوفة خاصة PRIVATE دون تأسيسها بقيمة ابتدائية ، فستؤسس كافة عناصرها بقيمة صفر NIL.

إضافة إلى طريقة الإعلان ، يمكن تجهيز متغيرات خاصة PRIVATE عندما:

- نمرر متغيرات إلى وظيفة باستخدام العبارة Parameter.



- نعين متغيراً دون إعلانه.

## مجال المتغيرات الخاصة PRIVATE

يبين المثال التالي مجال المتغير PRIVATE :

```
function main
myfunc1( )
? mvar // crasb-not visible here
return nil
```

```
function myfunc1
privat mave := 200 // note in-line initalization
myfunc2( )
? mvar
return nil
```

```
function myfunc2
mave = mvar * 2
myfunc3( )
return nil
```

```
function myfunc3
mvar = mvar + 16
return nil
```

تستدعي الوظيفة Main() الوظيفة MyFunc1() التي تعلن المتغير MVAR على أنه خاص private. ثم تستدعي الوظيفة MyFunc1() الوظيفة MyFunc2() التي تستدعي بدورها الوظيفة MyFunc3(). لاحظ أن MVAR لا يمرر كمتغير إلى الوظيفة MyFunc2() ولا الوظيفة MyFunc3() فهو مرئي في هاتين الوظيفتين اللتين تغيران قيمته.

عندما يعود التحكم إلى الوظيفة Main() يكون MVAR خارج المجال وبالتالي غير مرئي. ما أن المتغيرات مرئية ، يمكن إعلان المتغيرات الخاصة private في أحد الوظائف وتأسيسها في وظيفة ذات مستوى أدنى. لكن هذا النمط من البرمجة غير عملية وخطرة (ويجب تجنبها) لأنه من السهل جداً إحداث تغيير على المتغيرات وهذا يؤدي بدوره إلى

أخطاء أثناء التشغيل في جزء مختلف تماماً من برنامجنا. وقد يستغرق الأمر عدة ساعات لكشف هذه الأخطاء.

إن المتغير الخاص PRIVATE يبقى في مجاله إلى أن:

– نعود من الوظيفة التي أعلننا فيها المتغير لوظيفة بمستوى أعلى.

– نصدر أمر "مسح الجميع" Clear All

– نصدر أمر "مسح الذاكرة" Clear Memory

– نصدر أمر "تحرير" Release

يعتبر المتغير الخاص PRIVATE عبارة قابلة للتنفيذ ، ويجب أن يتبع أيّاً من عبارات "الحقل Field" و "MEMVAR" و "محلي LOCAL" و "ساكن STATIC".

يفضل تجنب استخدام المتغير الخاص PRIVATE مكان الإعلان عن المتغير المحلي LOCAL.

## إعلان المتغير العام PUBLIC

إن المتغيرات العامة PUBLIC موزونة لكافة الوظائف ضمن برنامجنا. وحالما نعلن عن المتغير العام PUBLIC فسيبقى كذلك دائماً طوال البرنامج.

إذا أعلننا عن متغير عام PUBLIC دون تأسيسه بقيمة ابتدائية ، فسيعين للمتغير قيمة البدء "غير حقيقي" (F.). ولكن إن أعلننا مصفوفة عامة PUBLIC array دون تأسيسها ، فستؤسس كافة عناصرها بقيمة الصفر NIL.

## مجال المتغير العام PUBLIC

يبين المثال التالي جمال المتغير العام PUBLIC :

```
function main
myfunc1( )
? mvar          // 416
? marray[1]     // "a"
return nil
```

```
function myfunc1
public mvar := 200
myfunc2( )
? mvar          // 416
? marray[1]     // "a"
return nil
```

```
function myfunc2
mvar = mvar + 2
myfunc3( )
return nil
```

```
function myfunc3
public marray[10]
mvar = mvar + 16
marray[1] = "a"
return nil
```

تستدعي الوظيفة ( ) Main الوظيفة ( ) MyFunc1 والتي تعلن عن المتغير MVAR بأنه عام PUBLIC. ثم تستدعي الوظيفة ( ) MyFunc1 الوظيفة ( ) MyFunc2 التي تستدعي بدورها على الوظيفة ( ) MyFunc3. ومع أن MVAR لا يمرر كمتغير إلى الوظيفة ( ) MyFunc2 أو الوظيفة ( ) MyFunc3 ، إلا أنه مرئي في هاتين الوظيفتين في المستوى الأدنى لأنه عام PUBLIC. وللسبب ذاته يكون MVAR مرئياً في الوظيفة ( ) Main.

تعلن الوظيفة ( ) MyFunc3 عن المصفوفة MARRAY بأنها من النوع العام PUBLIC لكنها لا تؤسسها بأية قيمة ابتدائية. ومع ذلك ، عندما نعلن عن المصفوفات في برنامج كليبر 5.2 فإن عناصره تؤسس ديناميكياً بقيمة الصفر NIL. عندما يعود تحكم البرنامج إلى الوظيفة ( ) MyFunc1 والوظيفة ( ) Main (اللتين تشيران

إلى عناصر في المصفوفة MARRAY تكون المصفوفة مرئية لأنه أعلن عنها بأنها عامة  
.PUBLIC

من الممكن تجاوز إعلاني خاص PRIVATE وعام PUBLIC. فعلى سبيل المثال إذا  
أعدنا كتابة الوظيفة ( ) MyFunc3 كما يلي:

```
function myfunc3
if mvar < 100
    public marray[10]
endif
mvar += 16
return nil
```

فلن يعلن عن المصفوفة MARRAY لأن MVAR يكون أكبر من ١٠٠ عندما يصل إلى  
هذا الوظيفة. وبالتالي سيتوقف برنامجنا ويتحطم عندما تحاول الوظيفة MyFunc1()  
الإشارة إلى MARRAY[1].

إن المتغير العام PUBLIC يبقى في مجاله إلى أن:

– نصدر أمر "مسح الجميع" Clear All

– نصدر أمر "مسح الذاكرة" Clear Memory

– نصدر أمر "تحرير" Release

كما هو الحال بالنسبة للمتغير الخاص PRIVATE ، فإن عبارات المتغير العام تعتبر  
عبارات قابلة للتنفيذ ، وبالتالي يجب أن يتبع أيّاً من عبارات "الحقل" "MEMVAR" و  
"محلي" و "ثابت".

لمتغير العام PUBLIC فوائد عديدة ، إلا أن كونه مرئياً في كافة أجزاء البرنامج  
يتعارض مع فكرة قابلية التركيب في برنامج كليبر 5.2. لذلك يفضل استخدام المتغير  
PUBLIC فقط عندما تكون هناك حاجة ملحة لاستخدامه.

لاحظ أن الحد الأعلى المصرح باستخدامه من المتغيرات الديناميكية في برنامج كليبر 5.2 هو (٢,٠٤٨) في أي وقت.

## إعلان المتغير المحلي LOCAL

تكون المتغيرات المحلية مرئية ضمن الوظيفة التي أعلنت فيها فقط. ومع أن المتغيرين المحلي LOCAL والخاص PRIVATE قد يبدو أن للوهلة الأولى متشابهين ، إلا أن الفارق الكبير بينهما هو أن المتغير المحلي LOCAL غير مرئي في الوظائف ذات المستويات الدنيا.

على غرار المتغير الخاص PRIVATE ، إذا أعلننا عن متغير محلي LOCAL ولم نؤسسه بقيمة ابتدائية ، فسيعين للمتغير قيمة البدء صفر NIL. وكذلك إذا أعلننا مصفوفة محلية دون تأسيسها ، فستؤسس كافة عناصرها بقيمة الصفر NIL.

بالإضافة إلى طريقة الإعلان ، يمكن تجهيز المتغيرات المحلية عندما نمررها إلى وظيفة ما باستخدام القاعدة اللغوية للقائمة LIST بدلاً من عبارة Parameter . وسيعامل المتغيران ( A ) و ( B ) في المثال التالي كمتغيرين "محليين".

```
function myfunc(a,b)
```

ويفضل أن تستقبل المتغيرات من خلال القائمة وليس من خلال العبارة .PARAMETER

## مجال المتغير المحلي LOCAL

يبين المثال التالي مجال المتغير المحلي LOCAL:

```
function myfunc1
  Local mvar := 200
  myfunc2()
  ? mvar // never gets this far
```

```
mvar
return nil
```

```
function myfunc2
  Mvar *= 2          // crasb-not visible here
return nil
```

وهو يوضح الفارق بين LOCAL و PRIVATE. تعلن الوظيفة ( ) MyFunc1 MVAR بأنه محلي LOCAL ثم تستدعي الوظيفة ( ) MyFunc2 التي تحاول تغيير قيمة MVAR ، ولكن بما أن هذا الأخير محلي LOCAL للوظيفة ( ) MyFunc1 فإنه غير مرئي للوظيفة ( ) MyFunc2 فيحدث خطأ أثناء التشغيل. والطريقة الأفضل هي أن نمرر المتغير من الوظيفة ( ) MyFunc1 إلى الوظيفة ( ) MyFunc2. إن أعلننا MVAR على أنه متغير خاص PRIVATE فسينفذ البرنامج ولكن قد تحدث أخطاء أخرى.

والحل الأمثل هو أن نمرر MVAR بالإشارة بحيث تعكس أية تغييرات تحدث في ( ) MyFunc2 في الوظيفة ذات المستوى الأعلى أيضاً.

```
function myfunc1
  local avar := 200
  myfunc2(@mvar)          // pass by referemce
  ? mvar                  // 400
return nil
```

```
function myfunc2(mvar)
  mvar *= 2
return nil
```

- عندما نعلن متغيراً محلياً LOCAL في وظيفة ما ، يجب أن يكون هذا الإعلان قبل أية عبارة قابلة للتنفيذ (بما في ذلك عبارات "خاص Private" و "عام Public" وحتى (Parameter).
- إن المتغيرات من النوع المحلي LOCAL تخفي المتغيرات العامة PUBLIC و المحلية LOCAL ، كما أنها تخفي أية حقول قاعدة بيانات لها الاسم ذاته.
- لا يمكن أن نستبدل ماكرو بالمتغير المحلي LOCAL لأن الأخير ليس له بنود في جدول الرموز .



- يجب أن نستخدم الوظيفة ( ) VALTYPE (كما بينا آنفاً) لفحص المتغير المحلي LOCAL. ويوجد في برنامج كليبر 5.2 أيضاً وظيفة ( ) TYPE لكنها تعمل فقط مع أنواع لها بنود في جدول الرموز. وبما أن المتغيرات المحلية LOCAL ليس لها بنود في جدول الرموز فلا فائدة من استخدام وظيفة ( ) TYPE معها.
- لا يمكن حفظ المتغيرات المحلية في ملفات الذاكرة (MEM). كما لا يمكن استرجاعها منها.

ملاحظة

من الضروري جداً استخدام المتغيرات المحلية LOCAL بدلا من الخاصة PRIVATE ، وبهذا سنجد أن برامجنا ستكون أسرع ، والوقت اللازم لصيانتها سيكون أقل.

## إعلان المتغير الساكن STATIC

تشبه المتغيرات الساكنة STATIC المتغيرات المحلية LOCAL في أنها مرئية ضمن الوظيفة الذي أعلنت فيها فقط. ومع ذلك فهي تمتاز عن المحلية بأنها تحتفظ بقيمتها طيلة مدة البرنامج. وفيما يلي توضيح لذلك:

```
function main
for x := 1 to 1000
  ? counter( )
next
return nil
```

```
function counter
static y := 1
return ++y
```

كلما نفذنا حلقة FOR..NEXT في وظيفة ( ) Main ، ستُراد القيمة المعادة من الوظيفة Counter( ) ، وبعبارة أخرى ، تكون قيمة (Y) تساوي واحد (١) في أول مرة تستدعى فيها وظيفة ( ) Counter ، ثم ستُراد مسبقاً إلى (٢) وهي القيمة المعادة من الوظيفة

( ) Counter. وفي المرة الثانية التي تستدعى فيها الوظيفة ( ) Counter ستحتفظ (Y) بقيمتها السابقة (٢) ، وتعيد الوظيفة القيمة (٣) ، وهكذا.

والسبب في ذلك أننا عندما نؤسس متغيراً ساكناً STATIC ، فسيعالج سطر الشيفرة هذا أثناء التجميع وليس أثناء التشغيل. وهذا معاكس تماماً للإعلانات العامة PUBLIC والخاصة PRIVATE والمحلي LOCAL التي تستوجب إعادة تأسيس المتغير كلما مررنا بهذه العبارة. إن أعلننا (Y) بأنه محلي أو خاص فسيعاد ضبط قيمته إلى (١) كلما استدعينا الوظيفة ( ) Counter التي سيعيد دائماً قيمة (٢) التي نريد.

وهذا يبدو عادياً لأننا اعتدنا أن ينفذ كل سطر من البرنامج كلما دخلنا الوظيفة. ولكن إذا دققنا في البرنامج السابق بواسطة برنامج كشف الأخطاء سنجد أن هذا الأخير يتجاوز الإعلان الساكن STATIC لأنه لا يعتبر عبارة قابلة للتنفيذ. إن أكثر ما يكون الإعلان الساكن STATIC مفيداً هو أثناء التجميع.

وعلى غرار المتغيرات "المحلية" و "الخاصة" ، إذا أعلننا متغيراً بأنه ساكن دون تأسيسه فسيعين للمتغير قيمة البدء صفر NIL. وكذلك ستضبط عناصر مصفوفة الساكن غير المؤسس على الصفر NIL ديناميكياً.

## مجال المتغير الساكن STATIC

يبين المثال التالي مجال المتغير الساكن STATIC:

```
function main static y := " testing"
for x := 1 to 100
  ? myfunc( )
  ?? y
next
return nil

function myfunc
static y := 100
return --y
```

تعلن الوظيفة ( ) Main المتغير (Y) بأنه ساكن STATIC وينفذ FOR..NEXT التي تستدعي وظيفة ( ) MyFunc. تعلن الوظيفة ( ) MyFunc أيضاً المتغير (Y) بأنه ساكن ويؤسسه إلى ١٠٠ أثناء التجميع. وفي كل مرة تُستدعى فيه الوظيفة ( ) MyFunc ستنقص قيمة (Y) وتعيد تلك القيمة. وبما أن المتغير (Y) ساكن STATIC فسيحافظ على قيمته في المرة التالية التي تستدعى فيها الوظيفة ( ) MyFunc. ومن الجدير ملاحظته بأن نسختي (Y) ستكونان مرئيتين ضمن الوظيفتين اللتين أعلنتا فيهما فقط.

- كما هو الحال في المتغيرات المحلية ، فإننا عندما نعلن متغيراً ساكناً في وظيفة ما ، يجب أن يكون هذا الإعلان قبل أية عبارة قابلة للتنفيذ ( بما في ذلك عبارات PRIVATE و PUBLIC و PRARMETER ). ويمكننا أيضاً إعلان المتغير الساكن STATIC قبل عبارة الوظيفة الأولى أو الإجراء الأول في ملف PRG. وسنبحث أدناه بشكل مختصر عن file-wide statics.
- للمتغيرات الساكنة أولوية عن المتغيرات "العامة" و "الخاصة" مثل المتغيرات المحلية وأيضاً عن أية حقول قاعدة بيانات لها الاسم ذاته.
- لا يمكن أن نستبدل ما كرو بالمتغير الساكن مثل المتغيرات المحلية لأن الأخير ليس له بنود في جدول الرموز.
- يجب أن نستخدم الوظيفة ( ) VALTYPE بدلاً من الوظيفة ( ) TYPE لفحص نوع متغير الساكن مثل المتغيرات المحلية.
- لا يمكن حفظ المتغيرات الساكنة في ملفات الذاكرة MEM. مثل المتغيرات المحلية كما لا يمكن استرجاعها منه.

## المتغيرات الساكنة على عرض الملف File-Wide Static Variables

إذا أعلننا متغيرات ساكنة STATIC قبل عبارة الوظيفة أو الإجراء ، فسيصبح مجال هذه المتغيرات عرض الملف وستكون مرئية في كافة وظائف ملفات البرنامج PRG. ، كما يمكننا أن نعتبر هذه المتغيرات الساكنة STATIC على عرض الملف وكأنها متغيرات عامة PUBLIC محدودة ، لأنها تشبه المتغيرات العامة لكنها خاصة بملف البرنامج PRG. ذاك فقط.

تمكّننا المتغيرات الساكنة على عرض الملف من كبسلة البيانات بتلك الوظائف التي نحتاج للوصول إليها فقط. وبكبسلة البيانات نصنع برنامجاً نموذجياً أكثر ونتخلص من الأخطاء المزعجة بواسطة إنقاص إمكانية الكتابة فوق المتغيرات المؤقتة.

### تحذير

إذا استخدمنا متغيرات ساكنة STATIC على عرض الملف ، فيجب أن نجمع ملف PRG. باستخدام خيار التجميع /N. فإذا لم نستخدم هذا الخيار فسيجهز برنامج كليبر إجراء بداية ضمن الملف PRG. مما سيجعل متغيراتنا الساكنة على عرض الملف عديمة الفائدة.

تأمل المثال التالي:

```
/* TEST.PRG */

static marray_ := { 'Greg', 'Jennifer', 'Justin' }
function func1
? marray_[1]
func2( )
return nil

function func2
? marray_[2]
func3( )
```

```
return nil
```

```
function func3  
? marray_[3]  
return nil
```

سيعمل هذا البرنامج بشكل تام إذا جمعنا الملف باستخدام الخيار /N ، ولكن إذا نسينا هذا الخيار فسيجهز برنامج كليبر 5.2 إجراء بداية ضمني يدعى "اختبار" Test. وسيقتصر مجال المصفوفة MARRAY\_ على هذا الإجراء الوهمي فقط.

## تأسيس المتغيرات الساكنة STATIC

يمكن تعيين المتغيرات الساكنة وقت إعلانها باستخدام معامل التعيين المباشر. ولكن لا يمكننا تعيينها إلا باستخدام ثوابت بسيطة فقط. ولايسمح باستدعاء الوظائف لأن المتغيرات الساكنة STATIC تؤسس قبل زمن التشغيل. وفي المثال التالي ، لايمكن تأسيس MVAR لأنه لايمكن تقييم وظيفة التاريخ ( DATE ) لأننا لم نشغل البرنامج بعد.

```
static mvar := date( )
```

إذا تطلب الأمر تعيين قيمة وظيفية ما إلى متغير ساكن STATIC ، فيجب علينا أن نعین هذه القيمة أثناء التشغيل وليس أثناء التجميع. وأسهل طريقة لعمل ذلك هي أن ندخل "اختبار الصفر" NIL Test لتحديد ما إذا كان المتغير الساكن قد أسس أم لا:

```
function counter  
static y  
if y == Nil  
  y := date( )  
endif  
return ++y
```







## اصطلاحات البرمجة Coding Conventions

تختلف طرق وأساليب البرمجة في جميع لغات البرمجة باختلاف المبرمجين. لذلك يفضل أن نختار أحد أساليب البرمجة التي نرتاح إليها ونلتزم بها. فإن تغيير الأسلوب بين الحين والآخر سيؤدي بنا إلى مشاكل عندما نحاول تعديل برامج قديمة.

### المساحة الفارغة White Space

يفضل دائماً استخدام مساحات فارغة قدر الإمكان بحيث يكون برنامج المصدر الذي أعدناه واضحاً للقراءة. ولن يكون للمساحة الفارغة أي تأثير على الأداء لأن مجمع برنامج كليبر 5.2 يتجاهلها.

### الإزاحة Indentation

يفضل إزاحة (ترك فراغ قبل) كافة كتل التحكم ، مع أن ذلك ليس واجباً. فإن ذلك يسهل علينا تحديد عبارة البداية (مثل IF و DO CASE و DO WHILE) التي تتطابق مع العبارة الموافقة ، خاصة عندما يكون لدينا تراكيب متداخلة. ويفضل ترك فراغ بقدر ثلاث مسافات بدلاً من استخدام هامش محدد مسبقاً ، رغم أن هذا عائد إلى المبرمج نفسه. يمكننا أيضاً اتباع أسلوب الإزاحة لكافة جسم الوظيفة كما في المثال المبين أدناه:

```
function test
  local x
  local y
  .
  .
return nil
```

## الأحرف الكبيرة Capitalization

يفضل استخدام الأحرف الكبيرة والصغيرة كما يلي: (أ) استخدام الأحرف الكبيرة مع كافة الكلمات الأساسية في برنامج كليبر 5.2. (ب) استخدام الأحرف الملائمة (كبيرة وصغيرة) مع الوظائف المحددة من قبل المستخدم (مثل: وظيفة ( MyFunc ). (ج) استخدام الأحرف الصغيرة في أسماء المتغيرات. وقد يفضل استخدام الأحرف الصغيرة في البرنامج كله ، باستثناء الأسماء في الثوابت الظاهرية التي يجب أن تكون دائماً بأحرف كبيرة ، مما يسهل علينا معرفة الثوابت الظاهرية بنظرة سريعة إلى البرنامج.

## أسماء المتغيرات Variable Names

ذكرنا في الفقرة السابقة أنه يفضل استخدام الأحرف الصغيرة في أسماء المتغيرات. لكن هذا يعرضنا خطأ إدخال رقم بدل حرف. لذلك يفضل البعض استخدام الحرف الأول من اسم المتغير لبيان نوعه (مثل: مصفوفة `a = array` و `b = code block` كتلة الشيفرة) كما في المثال التالي:

```
local cString := "a string"
local nAmount := 0
local IFlag := .f.
local oBrowse := tbrowsedb( )           // TBrowse object
local bCondition := { | | something }    // code block
local aChoices := { 1, 2, 3, 4 }         // array
```

لاحظ أن الحرف الثاني من اسم المتغير مكتوب بحرف كبير لتسهيل القراءة. ولاحظ أيضاً أن الأحرف العشرة الأولى فقط من كل اسم متغير مهمة.

## مختصرات الأوامر Command Abbreviations

مع أنه يمكن اختصار العديد من أوامر برنامج كليبر 5.2 ، باستخدام الأحرف الأربعة الأولى (وهذا يتوافق مع قاعدة البيانات dBASE III+) إلا أنه لا يفضل اختصار الأوامر لأن ذلك يجعل من الصعب قراءة برنامجنا.

## الإعلانات Declarations

يمكننا إعلان عدد من المتغيرات على سطر واحد كما يلي:

```
local a := 1, b := 2, c := 3, d := 4
```

ولكن يفضل كتابة إعلان كل متغير وتعيينه في سطر خاص:

```
local a := 1  
local b := 2  
local c := 3  
local d := 4
```

فهذا أسهل للقراءة ولن يؤثر على الأداء.

## الملاحظات Comments

ذكرنا آنفاً عدد من الأساليب المتوفرة في برنامج كليبر 5.2 لإضافة الملاحظات على برامجنا. ومع أن الأمر يتعلق بما يفضل البرمج ، إلا أنه يفضل استخدام المؤشر "//" للملاحظات المكونة من سطر واحد أو عدة أسطر. ويمكننا أيضاً إضافة الواصلة ( - ) أو خطوط أفقية ( آسكي 196) لفصل الملاحظة عن بقية البرنامج كما يلي:

```
// ----- here is the comment  
x := y + z
```

ويفضل استخدام التركيب "/\* \*/" مع الملاحظات متعددة الأسطر.



## أدوات واجهة المستخدم User Interface Tools

يوفر برنامج كليبر عدداً من الأوامر والوظائف سهلة الاستخدام للتحكم بما تعرضه الشاشة ومايفعله المستخدم. وسنبحث هذه الأوامر والوظائف بالتفصيل ضمن ثلاثة أقسام: الشاشة Screen ولوحة المفاتيح Keyboard وقوائم الاختيارات Menus.

### الشاشة Screen

وقبل أن ندخل في تفاصيل كل أمر ، يجب أن تعلم بأن صفوف وأعمدة الشاشة في برنامج كليبر تعتمد على الصفر وليس على الواحد. وبصورة أخرى ، الزاوية العلوية اليسرى هي الموضع (0.0) والزاوية السفلية اليمنى هي الموضع ( 24,79 ). ويلاحظ أيضاً أن العديد من هذه الأوامر ( SAY . و ؟ و ؟ ) يمكن أن تؤدي إلى طباعة البيانات على الطابعة وعرضها على الشاشة أيضاً.

#### أمر SAY..@

يعرض هذا الأمر البيانات أياً كان نوعها في موضع معين من السطر والعمود. وهو يعتمد على الجهاز ، أي إذا كان جهاز الإخراج الحالي هو الطابعة فسيؤدي إلى طباعة البيانات في موضع معين من السطر والعمود في الطابعة.

والقاعدة اللغوية هي:

@ <row> , <column> SAY <data> [COLOR <color>] [PICTURE <picture>]

إن < row> و < column> هما متغيران واضحان.

في حين أن <data> ويمكن أن تكون أي نوع من البيانات.

إذا استخدمنا فقرة اللون COLOR الاختيارية ، فسيتم عرض <data> بلون <color> المحدد. وهذا لن يغير ضبط الألوان الحالية (وسنبحث مسألة التحكم بالألوان بالتفصيل لاحقاً).

تمكنا الصورة PICTURE الاختيارية من صياغة البيانات بطرق متنوعة من خلال استخدام قالب الصورة أو وظيفة الصورة أو كليهما معاً. يحدد قالب الصورة طول البيانات التي ستعرض إضافة إلى قاعدة التنسيق لكل موضع. أما وظيفة الصورة فهي تنسيق للمخرجات ككل. وفيما يلي قائمة وظائف الصورة المتوفرة:

الوظيفة	التوضيح
B	عرض الأرقام محاذاة إلى جهة اليسار
C	عرض CR بعد الأرقام الموجبة الجدول مستمر من الصفحة السابقة....
D	عرض التاريخ في تنسيق SET DATE
E	عرض التاريخ والارقام بالتنسيق البريطاني
R	لحشر (إقحام) حروف بدون قالب
X	عرض DB بعد الأرقام السالبة
Z	عرض الأصفار كفراغات
(	يطوق الأرقام السالبة في هلالين
!	تحويل الحروف الأبجدية (الفبائية) إلى حروف كبيرة (لغة الإنجليزية)

إذا أردنا استخدام إحدى وظائف الصورة ، فيجب أن تبدأ سلسلة الصورة picture < > بالأمر "@" يليه مباشرة الحرف الموافق للوظيفة.

ونبين فيما يلي قائمة رموز القالب المتوفرة:

الرمز	الفعل
A, N, X, 9, #	عرض الأرقام لأي نوع من أنواع البيانات
L	عرض المنطقيات كـ: "T" و "F"
Y	عرض المنطقيات كـ: "Y" و "N"



الجدول مستمر من الصفحة السابقة...

الرمز	الفعل
!	تحويل الحروف الفبائية إلى الحروف الكبيرة (المجليزي)
\$	عرض اشارة الدولار بدلا من مسافة أمامية (رقمي)
*	عرض نجمة بدلا من مسافة أمامية
.	تحديد موقع النقطة العشرية

إذا عينا قالب صورة يحتوي أحرفاً غير مذكورة في القائمة السابقة (مثل الشرطة المعرّضة (-) في رقم هاتف) فإنها تنسخ حرفياً في المخرجات. وإذا أردنا إدراج هذه الأحرف في القائمة ، فيجب أن نستخدم وظيفة الصورة @R وإلا فستحل محل الأحرف الموافقة لقيمة العرض.

### تحذير

إذا أردنا استخدام وظيفة وقالب صورة معاً ، فإن الوظيفة تكون أولاً ، كما يجب فصلهما عن بعضهما بمسافة واحدة. وإن عدم فصلهما سيؤدي إلى ظهور أخطاء في برنامج كليبر.

فيما يلي بعض الأمثلة عن عبارة الصورة Picture:

```
@ 1, 1 say invoice->amount picture '#####.###'
@ 2, 1 say lineitem-> open picture 'Y'
@ 3, 1 say customer->phone picture '@R (###) ###-####'
@ 4, 1 say customer->last picture '@1'
@ 5, 1 say customer->first picture 'xxxxxxxxxxxxxx' // proper-case
```

### أمر @..Box

يعرض هذا الأمر مربعاً على الشاشة. وعلى عكس أمر SAY .. @ ، هذا الأمر خاص بالشاشة.

والقاعدة اللغوية هي:

@ <top>, <left>, <bottom>, <right> box <outline> [COLOR <color>]

<top> أعلى هو رقم يمثل الصف العلوي للمربع. و <left> يسار هو رقم يمثل أقصى عمود في يسار المربع. و <bottom> أسفل هو رقم يمثل الصف السفلي للمربع. و <right> يمين هو رقم يمثل أقصى عمود في يمين المربع.

<outline> الحد هو سلسلة حرفية تستخدم لتحديد شكل الإطار الخارجي للمربع. وقد يصل إلى ثمانية حروف طولاً ، والرموز الموافقة للمربع هي:

الرمز	الإطار الخارجي للمربع
1	الركن الأيسر العلوي
2	الحافة العلوية
3	الركن الأيمن العلوي
4	الجهة اليمنى
5	الركن الأيمن السفلي
6	الحافة السفلى
7	الركن الأيسر السفلي
8	الجهة اليسرى
9	رمز التعبئة

يشتمل برنامج كليبر 5.2 على ملف ترويسة يدعى ( BOX.CH ). يحتوي هذا الملف على مجموعة مفيدة من الثوابت الظاهرية manifest constants التي تحتوي على رموز آسكي ASCII الموسعة الضرورية والمناسبة لرسم حدود المربع.

الثابت الظاهري	التوضيح
B_SINGLE	مربع ذو حواف مفردة
B_DOUBLE	مربع ذو حواف مزدوجة
B_SINGLE_DOUBLE	مربع مفرد الحافة العلوية ، مزدوج الحواف الجانبية
B_DOUBLE_SINGLE	مربع مزدوج الحافة العلوية ، مفرد الحواف الجانبية

لاستخدام هذه الثوابت ، يجب أن نضمّن عبارة (`#include "box.ch"`) في برنامج المصدر الذي أعدناه قبل الإشارة إليها.

### ملاحظة

تحتوي هذه الثوابت على الرموز الثمانية فقط لحد المربع . فإذا أردنا تفريغ أو ملء الجزء الداخلي من المربع فيجب علينا أن نضيف الرمز التاسع.

على غرار SAY . . @ يمكننا اختيارياً أن نعين عبارة اللون color . فإذا عينت اللون فسيستخدم لعرض المربع.

وفيما يلي عدة أمثلة على أمر BOX . . @:

```
#include "box.ch"
```

```
@ 0, 0, maxrow( ), maxcol( ) box B_SINGLE+' ' // clear interior of box
@ 10, 10, 15, 60 box B_DOUBLE+' ' // clear interior of box
@ 5, 0, 10, maxcol( ) box B_SINGLE color 'w/r'
```

### الأمر ؟

يستطيع هذا الأمر أن يعرض بنداً أو أكثر من البيانات في السطر التالي للشاشة أو الطابعة (أو كليهما). والقاعدة اللغوية هي:

```
? [ <data> [ , <moredata> ... ] ]
```

يمكن أن تكون البيانات < data > من أي نوع. لاحظ أننا إذا عينا مصفوفة array أو هدفاً object ، أو كتلة شيفرة code block فستكون النتيجة أنه لن يطبع أي شيء.

أما إذا لم نعين متغيراً ، فسينقل الأمر (?) المؤشر (أو رأس الطابعة) إلى بداية الصف التالي.

## أمر ??

يشبه هذا الأمر أمر ( ? ) تماماً بفارق واحد هو أنه سيخرج البيانات في الموضع الحالي للمؤشر أو رأس الطابعة دون أن ينتقل المؤشر إلى بداية السطر التالي. والقاعدة اللغوية هي:

?? <data> [ , <moredata> ...]

وكما هو الحال في أمر (?) يمكن أن تكون البيانات < data > من أي نوع ، ولكن سيؤدي تعيين اسم المصفوفة أو الهدف إلى عدم طباعة أي شيء.

## الأمر CLS

لقد أصبحت الآن على دراية بطريقة عرض المعلومات على الشاشة ، وستجد بلا شك أن مسح الشاشة أمر ضروري. وفعلاً فإن الأمر CLS يقوم بهذا العمل تماماً ، باستخدام ضبط الألوان الحالي. لذلك ، إن كان اللون الحالي " W/ B " (الأبيض فوق الأزرق) فستصبح الشاشة بأكملها زرقاء.

## وظيفة الانزلاق ( ) SCROLL

إذا استدعينا هذا الوظيفة دون تمرير متغير لها ، فستُمسح الشاشة. فالفارق الوحيد بين هذه الوظيفة وأمر CLS هو أن الأمر CLS ينقل المؤشر إلى الصف "صفر" والعمود "صفر" بعد مسح الشاشة. لذلك يفضل استخدام الوظيفة ( ) SCROLL بدلاً من أمر CLS إذا كنا لا نريد تغيير موضع المؤشر.

## أمر المسح @.Clear

يمكننا هذا الأمر من مسح أجزاء من الشاشة. والقاعدة اللغوية هي:

@ <top>, <left> CLEAR [TO <bottom> , <right> ]

يمثل < top > الصف العلوي و < left > العمود الأيسر للمنطقة التي ستمسح. فإذا لم نحدد < bottom > و < right > فستمسح كافة أجزاء الشاشة التي تحت الموضع العلوي الأيسر. وكما هو الحال مع أمر CLS ووظيفة ( SCROLL ) ، يستخدم ضبط الألوان الحالي.

## التحكم بالألوان Color Control

يمكننا الأوامر (@.SAY) و (@.BOX) من تعيين اللون الذي نريد عرض البيانات أو المربع به. (ويمكننا أيضاً تغيير اللون الافتراضي في أي وقت باستخدام وظيفة ضبط الألوان ( Setcolor ). وعند تعيين ألوان هذين الأمرين ، يجب أن نستخدم القاعدة اللغوية "<foreground>/<background>" واللتين تعنيان على التوالي "اللون الأمامي/ اللون الخلفي" ويمكنك القيام بتعيين اللونين من الجدول التالي:

اللون	حرف الشيفرة	المكافئ في الشاشة أحادية اللون
أسود	N	أسود
أزرق	B	تحت خط
أخضر	G	أبيض
أزرق داكن	BG	أبيض
أحمر	R	أبيض
فوشي (أحمر مزرق)	RB	أبيض
بنّي	GR	أبيض
أبيض	W	أبيض

الجدول مستمر من الصفحة السابقة....

اللون	حرف الشيفرة	المكافئ في الشاشة أحادية اللون
رمادي	N+	أسود
أزرق لامع	B+	فاتح تحته خط
أخضر لامع	G+	أبيض لامع
أزرق داكن لامع	BG+	أبيض لامع
أحمر لامع	R+	أبيض لامع
فوشي لامع	RB+	أبيض لامع
أصفر	GR+	أبيض لامع
أبيض لامع	W+	أبيض لامع
أسود	U	تحته خط
فديو معكوس	I	فديو معكوس
فارغ	X	فارغ

يمكننا تعيين الحروف باستخدام حروف صغيرة أو كبيرة. كما يمكننا إضافة نجمة ("\*) إلى أي لون أمامي لنجعله يومض blink. كما يمكن أن يؤدي استخدام النجمة مع وظيفة ضبط الوميض (Setblink) إلى تحسين ألوان الخلفية.

فيما يلي أمثلة عن عدة أزواج من الألوان:

```
setcolor("+W/R") // bright white on red
@ 1,1 say "hello" color "n/b" // black on blue
@ 0,0,10,20 box B_SINGLE+' ' color "+GR/G" // yellow on green
@ 20,10 say "WARNING!" color "w/r" // blinking white on red
```

## لوحة المفاتيح Keyboard

فيما يلي الوظائف والأوامر التي يمكننا من التفاعل مع المستخدم في برنامج كليبر 5.2.



## وظيفة مفتاح الإدخال ( ) INKEY

تفحص هذه الوظيفة الذاكرة المؤقتة buffer للوحة المفاتيح بحثاً عن مفتاح إدخال. فإن وجد ، فستعيد قيمة رقمية ما بين (47 -) و (419). فإن لم يجد أي مفتاح إدخال ، فستعيد الوظيفة ( ) INKEY قيمة الصفر Zero.

يوجد في دليل توثيق برنامج كليبر 5.2 المطبوع وفي ملف "دليل نورتن" Norton Guide قائمة كاملة بقيم وظيفة ( ) inkey. إضافة إلى ذلك ، يحتوي ملف الترويسة ( INKEY.CH ) المتوفر في كليبر 5.2 كشفاً بالشواهد الخاصة بكل مفتاح تقريباً. ولاستخدام هذه الشواهد يجب علينا أن نكتب عبارة ( #include "inkey.ch" ) في برنامج المصدر الذي أعدناه قبل الإشارة إليها.

يمكننا اختياريًا تمرير متغير رقمي للوظيفة ( ) inkey ، لتحديد عدد الثواني التي ينتظر فيها الوظيفة ضغطة مفتاح. وإن مررنا الصفر متغير (أي inkey(0)) فإن البرنامج سيتوقف إلى أن يضغط المستخدم أحد المفاتيح. وهذه أفضل طريقة لحث المستخدم على ضغط المفاتيح كما سنرى في المثال التالي. لاحظ استخدام وظيفة ( ) CHR التي تحول القيمة الرقمية للوظيفة ( ) inkey إلى الحرف أو الرمز الذي يعادها.

```
? "press 'Q' to quit"
key := inkey(0)
if upper (chr(key) ) $ "Q"
    quit
endif
```

سنستخدم في المثال التالي وظيفة ( ) inkey لعرض رسالة على الشاشة لمدة ١٠ ثوان أو إلى أن يضغط المستخدم أحد المفاتيح (أيهما يكون الأول).

```
@ 12, 16 say "MARNING: system will shut down in five minutes!"
inkey(10)
cls
```

السيئة الوحيدة لوظيفة inkey( ) هو أنها لا تعتبر حالة إنتظار أصلية. وبعبارة أخرى ، فإنها لا تأخذ " مفاتيح الاستخدام السريع " hot keys بعين الاعتبار (راجع وظيفة ضبط المفاتيح ( Setkey أدناه). ومع ذلك ستحول في المثال التالي وظيفة inkey( ) إلى حالة إنتظار. وبدلاً من استدعاء الوظيفة inkey(0) في برنامجنا ، سنستدعي الوظيفة Myinkey( )

```
function myinkey
local key := inkey(0)
local block := setkey(key)
if block <> NIL // there is a code block for this key
    eval(block, procname(1), procline(1), 'myinkey')
endif
return key
```

## وظيفة آخر مفتاح LASTKEY( )

كما هو واضح من اسمه ، تعيد هذه الوظيفة القيمة الرقمية لآخر مفتاح ضغط. ويمكن أن تستخدم هذا الوظيفة في التفريع الشرطي ، مثل التأكد من أن المستخدم لم يضغط مفتاح الخروج ESC للخروج من شاشة إدخال البيانات. في كل مرة نستدعي الوظيفة INKEY( ) فستُحدَّث الوظيفة Lastkey( )

## وظيفة المفتاح التالي Nextkey( )

تشبه هذه الوظيفة وظيفة inkey( ) في أنها تفحص الذاكرة المؤقتة للوحة المفاتيح بحثاً عن مفتاح إدخال وتعيد القيمة الرقمية. لكن الوظيفة Nextkey( ) لا تستخرج المفتاح من الذاكرة المؤقتة ولا تُحدِّث قيمة الوظيفة Lastkey( ) . كما أنها لا تقبل أية متغيرات من النوع parameter.

## وظيفة التنبيه ( ) ALERT

هذه الوظيفة مفيدة للمستخدم ، فهي تترك البيئة الحالية دون أن تؤثر فيها أبداً. بل إنها تحفظ قيمة آخر مفتاح في الذاكرة المؤقتة للوحة المفاتيح (المعادة من قبل الوظيفة (Lastkey( ).

والقاعدة اللغوية هي:

ALERT( <cMessage> [, <aOptions> ] [, <cColor> ] )

الرسالة <cMessage> هي التي ستعرض أمام المستخدم. يمكن أن تشتمل هذه الرسالة على أكثر من سطر واحد. ويتم الفصل بين السطور بفاصلة منقوطة (;).

أما الخيار <aOptions> فهو مصفوفة من الخيارات التي يجب على المستخدم أن يختار أحدها ، مثل [ "إنهاء" Quit ، "إعادة المحاولة" Retry ]. إن لم يمرر هذا المتغير فسيستخدم خياراً واحداً هو ("موافق" OK).

المتغير غير المذكور في دليل التوثيق <cColor> هو سلسلة حرفية تتحكم باللون الذي يعرض به المربع. ويفضل ألا نعتمد عليه لأن جميع الميزات غير الموثقة قد تحذف في إصدارات لاحقة لبرنامج كليبر 5.2.

تقوم وظيفة التنبيه ( ) ALERT برسم مربع وتعرض الخيارات كأزرار. ثم تنتظر أن يضغط المستخدم أحد أحرف الاختيار ، وتعيد الرقم الموافق للخيار الذي اختاره المستخدم. فإذا ضغط المستخدم مفتاح الخروج [Esc] ، فستعيد الوظيفة ALERT() القيمة صفر NIL.

يوضح المثال التالي استخدام هذه الوظيفة:

```
if lastkey( ) == k_.and. alert("Are you want to quit", ;  
    { "Quit", "Continue" } ) == 1
```

```
quit
endif
```

## الأمر @.GET

يضع هذا الأمر بند البيانات في موضع محدد على الشاشة لإدخال البيانات على الشاشة، ويؤسس الهدف GET لإدخال البيانات. والقاعدة اللغوية هي:

```
@ <row> , <column> GET <var> [ picture <pic> ] ;
[ COLOR <color> ] [ WHEN <when> ] [ VALID <valid> ] ;
```

إن الصف < row > و العمود < column > يحددان موقع عرض الهدف GET على الشاشة.

كما تعمل عبارة الصورة Picture الاختيارية مع هذا الأمر بشكل مشابه للأمر @.SAY. وعلاوة على الخيارات المتوفرة لأمر @.SAY، يمكننا أيضاً استخدام وظيفة الصورة @Snn والوظيفة @K. حيث تمكّننا الوظيفة @S من عرض جزء من حرفية GET الطويلة فقط. والانزلاق خلالها عندما يقترّب المستخدم من النهاية. تمثل (nn) الطول المحجوز للهدف GET. أما الوظيفة @K فتؤدي إلى مسح المحتويات السابقة للهدف GET حالما يضغط المستخدم أحد المفاتيح.

تختلف القاعدة اللغوية لعبارة اللون Color الاختيارية مع هذا الأمر عنها مع الأمرين @.SAY و @.BOX. فإذا استخدمنا هذه العبارة، فيجب أن يكون ضبط الألوان بصيغة " <enhanced> , <standard> " قياسي ومحسن. فتستخدم زوج الألوان "المحسن" عند تظليل الهدف GET، أما زوج الألوان "القياسي" فيستخدم عندما لا يظلل. وعلى سبيل المثال، إذا أردنا أن يكون الهدف GET بلون أبيض على خلفية حمراء عندما يظلل، وأبيض على خلفية زرقاء عندما لا يظلل، يمكننا أن نستخدم العبارة التالية:

```
@ 20, 0 get x color "w/b , w/r"
```

تستخدم العبارتان "WHEN" و "VALID" للتدقيق السابق واللاحق على التوالي وسنبحثهما بالتفصيل فيما يلي.

تستخدم عبارة "أرسل" Send الاختيارية لإرسال رسالة < message > إلى هدف GET. وقد تم بحث هذه الميزة وميزات أخرى عند مناقشة تصنيفات هدف كليبر 5.2 في جزء آخر من الكتاب: "تصنيفات هدف كليبر 5.2".

## أمر القراءة READ

ينشط هذا الأمر أية أهداف GET. @ معلقة بحيث يمكن للمستخدم أن يدخل فيها بيانات. كما أنه يسمح بأهداف GET المعلقة حالما تنتهي منها مباشرة. المفاتيح النشطة ضمن أمر القراءة هي:

المفتاح	الوظيفة
	الانتقال إلى اليسار حرف واحد داخل GET
	الانتقال إلى اليمين حرف واحد داخل GET
	الانتقال إلى اليسار كلمة واحدة داخل GET
	الانتقال إلى اليمين كلمة واحدة داخل GET
	الانتقال إلى GET السابقة
	الانتقال إلى GET التالية
	الانتقال إلى GET التالية (أو الانهاء إذا كان على GET الأخيرة)
	الانتقال إلى الحرف الأول في GET
	الانتقال إلى الحرف الأخير من GET

الجدول مستمر من الصفحة السابقة

المفتاح	الوظيفة
Ctrl - Home	الانتقال إلى بداية أول GET
Ctrl - End	الانتقال إلى بداية آخر GET
Delete	حذف حرف من موقع المؤشر
BkSp	حذف الحرف الحالي والرجوع إلى اليسار
Ctrl - T	حذف كلمة من اليمين
Ctrl - Y	حذف من بداية موقع المؤشر إلى نهاية GET
Ctrl - U	استرداد GET الحالية إلى قيمتها الأصلية
Insert	التبديل بين الاقحام (الحشر) و الكتابة الفوقية
Ctrl - W Ctrl - C	إنهاء عملية القراءة READ ، وحفظ GET الحالية
PgUp PgDn	إنهاء عملية القراءة READ ، وحفظ GET الحالية
Esc	إنهاء عملية القراءة ، دون حفظ GET الحالية

## تدقيق البيانات المدخلة Validating Data Entry

يوفر برنامج كليبر 5.2 عبارتي "WHEN" و "VALID" اللتين تمكنا من إجراء التدقيق السابق والتدقيق اللاحق لكافة البيانات المدخلة. وكلاهما تقبل أية عبارة من كليبر تكون نتيجة تقييمها عبارة منطقية. وهذا يشمل مقارنة بسيطة ، أو إشارة إلى متغير منطقي أو حقل قاعدة بيانات ، أو استدعاء وظيفة.



## عبارة WHEN

يتم تقييم عبارة WHEN قبل أن ندخل هدف GET. فإن كانت النتيجة "غير صحيح" (F.) فستمنعنا من إدخال هدف GET أما عبارة VALID فيتم تقييمها عندما نحاول الخروج من هدف GET. فإن كانت النتيجة "غير حقيقي" (F.) فلن نستطيع الخروج من هدف GET.

بين المثال التالي أبسط استخدام لعبارة WHEN. لن يتمكن المستخدم من إدخال رقم بطاقة الائتمان ما لم يضبط متغير رصيد الدائن إلى "حقيقي" (T.).

```
@ 12, 20 say "Credit? " get credit
@ 13, 20 say "Card Number: " get cardno when credit
```

توفر عبارة WHEN إمكانيات عديدة باستدعاء وظائف منها. وما علينا إلا التأكد من أن عبارة WHEN تقييم بقيمة منطقية. (يجب أن تكون هذه القيمة "حقيقي" (T.) إن أردنا إدخال هدف GET).

يبين المثال التالي كيفية إعداد رسالة لكل هدف GET.

```
@ 10,0 get name when fieldhelp(24, 1, "Please enter a name")
@ 11,0 get address when fieldhelp(24, 1, "Please enter an address")
@ 12,0 get city when fieldhelp(24, 1, "Please enter a city")
read
```

```
function fieldhelp(row, col, msg)
@ row, col say padr(msg, 50)
return .t.
```

## شرط VALID

يتم تقييم الشرط VALID ، كما ذكرنا أعلاه ، عندما نحاول الخروج من هدف GET. فإذا كانت النتيجة "غير حقيقي" (F.) فلن نستطيع الخروج من هدف GET.

يبين المثال التالي أبسط صيغة للتدقيق. إنها تجبر المستخدم على إدخال رقم أكبر من الصفر.

```
@ 1, 1 get x valid x > 0
```

سنستدعي في المثال التالي وظيفة تدقيق تنفذ "بحث" SEEK في قاعدة بيانات. (ويفترض وجود فهرس تحكم في قاعدة البيانات هذه) فإذا لم توجد البيانات في قاعدة البيانات ستستخدم وظيفة التنبيه (ALERT) لعرض رسالة خطأ ، وتعيد وظيفة التدقيق "غير حقيقي" (F.) ، وبالتالي إجبار المستخدم على إدخال بيانات أخرى.

```
@ 1, 1 get cName valid checkname(cName)
read
.
.
function checkname(cName)
local IFound := .t.
if ! lookup->( dbseek(cName) )
    alert(cName + " is not a valid entry!")
    IFound := .f.
endif
return IFound
```

نستخدم في المثال التالي قاعدة منطقية مشابهة للتأكد من أن المستخدم لم يدخل قيمة مفتاح متكرر.

```
@ 1, 1 get cName valid checkname(cName)
read
.
.
function checkname(cName)
local IFound := .f.
if ! lookup->( dbseek(cName) )
    alert(cName + " is not a valid entry!")
    IFound := .t.
```

```
endif  
return .not. IFound
```

## عمليات القراءة المتداخلة Nested READs

من السهل إجراء عمليات قراءة متداخلة في مستويات مختلفة من أهداف GET في برنامج كليبر 5.2 ، بما أن أهداف GET تحمل بشكل مصفوفات GETLIST ، فكل ما علينا عمله هو التأكد من استخدام المصفوفة LOCAL GETLIST في الوظيفة التي تنفذ القراءة المتداخلة وهذا سيحمي مصفوفات GETLIST التي هي بمستوى أعلى من أن تمسح بعد أمر القراءة READ. كما ينبغي عدم استخدام أوامر "مسح أهداف" CLEAR GETS لأنها ستسمح كافة مستويات أهداف GET.

سنستخدم في المثال التالي القراءة المتداخلة في وظيفة صحيحة ، والتي تستدعي من الهدف GET التالي.

```
1 function main  
2 local a := 0  
3 local b := 0  
4 local c := 0  
5 cls  
6 @ 1,1 get a  
7 @ 2,1 get b valid nestread( )  
8 @ 3,1 get c  
9 read  
10 return nil  
11  
12 function nestread  
13 local d := 1  
14 // for a nested read, all you need to do is put the following line  
15 // in the function where the nested read will take place (and be  
16 // sure not to use the CLEAR GETS command)  
17 local getlist := { }  
18 @ 5, 1 say "In nested read . . ."  
19 @ 6, 1 get d  
20 read  
21 @ 5, 0 clear  
22 return .t.
```

## أمر ضبط المفاتيح SET KEY

يضيف هذا الأمر بعداً جديداً إلى برنامجنا. ويمكننا استخدامه لتجهيز "مفاتيح الاستخدام السريع" "hot Keys" التي تنفذ وظائف غير مرتبطة بالقواعد المنطقية للبرنامج التقليدي. وكمثال على استخدام مفاتيح الاستخدام السريع هو تشكيل مفتاح (F1) لعرض "مساعدة خاصة بالسياق" للحالة الراهنة للبرنامج.

القاعدة اللغوية لأمر ضبط المفاتيح هي:

SET KEY <key> TO <action>

<Key> هو تعبير رقمي يمثل قيمة الوظيفة ( INKEY ) للمفتاح المطلوب. يمكننا استخدام ملف ( INKEY.CH ) الذي يحتوي جميع قيم وظيفة ( INKEY ) لاستخدام إحدى القيم. وأما الإجراء < action > فهو اسم الإجراء أو الوظيفة التي ستنفذ عندما يضغط المستخدم المفتاح < Key > ضمن حالة الانتظار في برنامج كليبر.

عندما نستخدم أمر ضبط المفاتيح SET KEY ، ستمرر ثلاثة متغيرات آلياً إلى إجراء أو وظيفة مفتاح الاستخدام السريع الذي إختارناه. وهذه المتغيرات هي: اسم الإجراء الحالي ، ورقم السطر الحالي لبرنامج المصدر ، واسم المتغير الذي يُقرأ. ويمكن استخدام اسمي الإجراء والمتغير لتجهيز نظام مساعدة ، أما رقم السطر فيتغير بسرعة ولا يمكن الاعتماد عليه.

كما يمكن استخدام الوظيفة ( SETKEY ) لفحص المفتاح وتحديد ما إذا كان "المفتاح سريع الاستخدام" أم لا . فإن كان كذلك احفظ حالته واستعدّها.

## قوائم الاختيارات Menus

يوفر برنامج كليبر 5.2 عدة وسائل لتجهيز قوائم الاختيارات تعرض على الشاشة بشكل مضيء.

### أمر "التوجيه" @..PROMPT

يؤسس هذا الأمر ويعرض خياراً من قائمة الاختيارات بشكل مضيء. والقاعدة اللغوية هي:

@ <row> , <column> PROMPT <prompt> [ MESSAGE <message> ]

الصف <row> و العمود <column> هما عبارتان رقميتان تحددان مكان عرض الخيار على الشاشة.

التوجيه <prompt> هو تعبير حرفي سيستخدم كنص للخيار.

وأما المتغير الاختياري فهو رسالة <message> هو تعبير حرفي يُعرض في صف الرسالة عندما يظلل هذا الخيار . ويحدد أمر "ضبط الرسالة" SET Message صف الرسالة وهو السطر رقم ٢٤ كما تعلم. ويمكننا أيضاً وبشكل اختياري تعيين الرسالة <message> لبعض خيارات قائمة الخيارات وليس كلها.

لاحظ أن ترتيب أوامر @..PROMPT يؤثر بشكل مباشر على ترتيب الخيارات ضمن قائمة الاختيارات المعروضة بشكل مضيء. سنناقش هذا الموضوع بالتفصيل عند الحديث عن قائمة الخيارات MENU TO إن شاء الله.

موضع المؤشر : كلما أصدرنا أمر @..PROMPT يعاد ضبط موقع المؤشر على الشاشة إلى ما وراء أقصى حرف في يمين الخيار مباشرة. ويمكننا تعديل ذلك باستخدام وظيفتي الصف ( ) ROW و العمود ( ) COL لوضع المؤشر في مكان ملائم للخيارات اللاحقة كما سيوضح في المثال أدناه.

```
@ 10, 0      prompt "Option 1"
@ row( ), col( ) + 2 prompt "Option 2"
@ row( ) + 1, 0      prompt "Option 3"
@ row( ), col( ) + 2 prompt "Option 4"
```

الفكرة الرئيسة هي أن نعرض كافة خيارات قائمة الاختيارات بواسطة أمر @..PROMPT ثم نستخدم أمر الاختيار من قائمة الاختيارات MENU TO لتشغيل قائمة الاختيارات المعروضة بشكل مضىء.

## أمر الاختيار من قائمة الاختيارات MENU TO

يشغل هذا الأمر عملية تظليل وارتداد قائمة الاختيارات المعروضة بشكل مضىء والقاعدة اللغوية له هي :

MENU TO <var>

<var> هو اسم المتغير أو عنصر المصفوفة الذي سيعين له قيمة رقمية حسب الخيار الذي يختاره المستخدم. فإذا عينا قيمة رقمية لـ: <var> قبل عبارة MENU TO ، فستستخدم هذه القيمة لتحديد الخيار الذي يجب تظليله وإلا فيسظلل الخيار الأول.

يمكن تشغيل المفاتيح التالية في قائمة الخيارات ذات الشريط المضىء light-bar menu:



المفتاح	الوظيفة
↑	الانتقال إلى خيار القائمة السابق - إذا كان على أول خيار و الالتفاف ممكن ، يتم الانتقال إلى آخر خيار في القائمة. (انظر SET WRAP)
↓	الانتقال إلى خيار القائمة التالي - إذا كان على آخر خيار في القائمة ووظيفة الالتفاف ممكنة، يتم الانتقال إلى أول خيار في القائمة. (انظر SET WRAP)
←	الانتقال إلى خيار القائمة السابق - إذا كان على أول خيار و الالتفاف ممكن ، يتم الانتقال إلى آخر خيار في القائمة. (انظر SET WRAP)
→	الانتقال إلى خيار القائمة التالي - إذا كان على آخر خيار في القائمة ووظيفة الالتفاف ممكنة، يتم الانتقال إلى أول خيار في القائمة. (انظر SET WRAP)
Home	الانتقال إلى أول خيار في القائمة
End	الانتقال إلى الخيار الأخير في القائمة
Enter	اختيار الخيار الحالي والخروج
PgUp	اختيار الخيار الحالي والخروج
PgDn	اختيار الخيار الحالي والخروج
Esc	الرجوع إلى الصفر والخروج

يمكننا أيضاً اختيار أحد الخيارات بضغط المفتاح الموافق للحرف الأول منه. فمثلاً، يمكن اختيار خيار التقارير Reports بضغط مفتاح "R" أو "r".

يعرض الخيار المختار (المظلل) باللون المحسن الحالي. أما باقي الخيارات غير المختارة فتعرض باللون القياسي الحالي.

يتحدد ترتيب الخيارات في قائمة الاختيارات وفق ترتيب أوامر PROMPT..@. فبان رتبنا الخيارات كما يلي:

```
@ 20, 0 prompt "option 1"
@ 19, 0 prompt "option 2"
@ 18, 0 prompt "option 3"
@ 17, 0 prompt "option 4"
menu to sel
```

فإن الخيار الأخير سيعتبر الأول ، والخيار الأول سيعتبر الأخير. ويمكن استخدام مفتاحي السهمين إلى أعلى و إلى أسفل للانتقال إلى أسفل القائمة ، مما قد يزعج المستخدم. لكن هذه المعلومات تفيدنا في معرفة الإمكانيات المتاحة لنا فمثلاً ، يمكننا ترتيب الخيارات في قائمة الاختيارات من اليسار إلى اليمين ومن أعلى إلى أسفل كما يلي:

```
@ 10, 0 prompt "option 1"
@ 10, 15 prompt "option 2"
@ 11, 0 prompt "option 3"
@ 11, 15 prompt "option 4"
menu to sel
```

### أمر ضبط الرسالة SET MESSAGE

يحدد هذا الأمر ، كما ذكرنا أعلاه ، مكان عرض الرسالة على الشاشة. والقاعدة اللغوية هي:

SET MESSAGE TO <row> [CENTER]

فالصف < row> هو تعبير رقمي يمثل الصف الذي ستعرض عليه الرسالة الخاصة بالخيار المظلل.

فإذا كتبنا كلمة وسط Center بعد الصف < row> فستعرض الرسالة آلياً في وسط الصف المطلوب.

ولا تعرض الرسالة إلا إذا أصدرنا أمر ضبط الرسالة SET MESSAGE.

### أمر ضبط اللف SET WRAP

يمكننا هذا الأمر من اللف من أعلى إلى أسفل وبالعكس. وإن اللف يوقف تشغيله افتراضياً ما لم نصدر هذا الأمر. والقاعدة اللغوية هي:

SET WRAP ON / OFF

يمكننا أيضاً تمثيل عبارة منطقية بدلاً من كلمة "ON" أو "OFF". فالعبارة المنطقية "حقيقي" (.T.) تعادل "ON"، و "غير حقيقي" (.F.) تعادل "OFF". وإن استخدمت هذه الطريقة فيجب أن نضع العبارة ضمن قوسين كما يلي:

```
lWrap = .T.  
set wrap (lwrap)
```

## وظيفة الاختيار ACHOICE()

تمكّننا هذه الوظيفة من عرض مصفوفة سلاسل حرفية على الشاشة ليختار منها المستخدم. والقاعدة اللغوية هي:

```
ACHOICE( <top>, <left>, <bottom>, <right>, <array> ;  
         [ , <selectable> ] [ , <user-function> ] ;  
         [ , <initial> ] [ , <window row> ] )
```

المتغيرات < top> و < left> وأسفل < bottom> ويمين < right> تحدد منطقة الشاشة التي ستعرض فيها قائمة الاختيارات. يجب أن تكون هذه المنطقة عريضة بشكل كافٍ لتسع لأعرض سلسلة حرفية في المصفوفة، وإلا فستقطع هذه السلسلة. ولكن ليس من الضروري أن تكون هذه المنطقة طويلة بحيث تتسع لطول المصفوفة بأكمله، لأن وظيفة ACHOICE() تعرض البنود حسب حركتنا على الشاشة إلى أعلى أو إلى أسفل.

لاحظ أن وظيفة ACHOICE() لا يكون له إطار تلقائياً، بل يجب علينا رسم هذا الإطار (وهذا أمر سهل).

المصفوفة <array> هي مصفوفة مكونة من سلاسل حرفية.

المتغير الاختياري، قابل للاختيار <selectable> يمكن أن يكون أحد شيئين:

• الأول أن نجعله مصفوفة تحتوي قيماً منطقية. ويجب أن يكون طوله مثل طول المصفوفة <array> وأن تتوافق كل قيمة منطقية مباشرة مع العنصر ذاته في <array>. تشير القيمة "حقيقي" (T.) بأن العنصر المقابل قابل للاختيار بينما تعين القيمة "غير حقيقي" (F.) أن العنصر المقابل غير قابل للاختيار وسنتجاوزه.

• الثاني أن نجعله قيمة منطقية. القيمة "حقيقي" (T.) تعني أن كافة العناصر قابلة للاختيار ، و القيمة "غير حقيقي" (F.) تعني أنه لا توجد عناصر قابلة للاختيار (يفيدنا الخيار الثاني في الحالات التي نريد فيها عرض محتويات المصفوفة دون السماح بالاختيار).

المتغير الاختياري وهو وظيفة معرفة من قبل المستخدم <user-function> تمكّننا من تعيين وظيفة يحددها المستخدم ويمكن أن يعالج ضغوطات المفاتيح.

المتغير الاختياري مبدئي <initial> يمكننا من تعيين عنصر المصفوفة التي يجب أن تظل مبدئياً. العنصر الأول في المصفوفة يظل بشكل افتراضي.

المتغير الاختياري صف النافذة <window row> يمكننا من تعيين صف النافذة الذي سيظهر عليه العنصر المظلل مبدئياً. ترقيم الصفوف ابتداءً بالصف من أعلى <top>. وبشكل افتراضي ، سيستخدم صف النافذة الأول (أو الأعلى).

عندما يختار المستخدم ، ترجع الوظيفة ( ACHOICE ) رقم العنصر المختار. وإذا مر المستخدم قيمة مفتاح الخروج [Esc] للخروج ، فترجع الوظيفة قيمة الصفر zero. يجب أن نفحص البرنامج بحثاً عن حالة الخروج بدلاً من افتراض أن القيمة المرجعة ستكون رمزاً سلبياً صحيحاً للمصفوفة. ويوضح المثال التالي ذلك:

```
// bad
x := achoice(0, 0, 10, 50, array)
? array[x] // crashes if user escaped from ACHOICR( )
// good
x := achoice(0, 0, 10, 50, array)
if x > 0 // make certain user didn't escape
  ? array[x]
endif
```

تتغير وظيفة ( ) ACHOICE حسب المفاتيح العاملة فيه. فإذا كنا لانستخدم المتغير <user-function> فستكون المفاتيح التالية عاملة:

المفتاح	الوظيفة
↑	الانتقال إلى أعلى فقرة واحدة
↓	الانتقال إلى الأسفل فقرة واحدة
←	الخروج دون اختيار
→	الخروج دون اختيار
Home	القفز إلى العنصر الأول في المنظومة
End	القفز إلى العنصر الأخير في المنظومة
Ctrl - Home	القفز إلى العنصر الأول في النافذة
Ctrl - End	القفز إلى العنصر الأخير في النافذة
PgUp	الانتقال إلى الصفحة السابقة
PgDn	الانتقال إلى الصفحة التالية
Ctrl - PgUp	القفز إلى أول عنصر في المنظومة
Ctrl - PgDn	القفز إلى آخر عنصر في المنظومة
Enter	انتقاء الفقرة الحالية والخروج
Esc	الخروج دون اختيار
الحروف	القفز إلى الفقرة التالية يبدأ بذلك الحرف

وإذا كنا نستخدم المتغير <user-function> فستكون المفاتيح التالية فقط هي العاملة:

المفتاح	الوظيفة
↑	الانتقال إلى أعلى بند واحد
↓	الانتقال إلى أسفل بند واحد
Ctrl - Home	القفز إلى البند الأول في النافذة
Ctrl - End	القفز إلى البند الأخير في النافذة
PgUp	الانتقال إلى الصفحة السابقة
	الجدول مستمر من الصفحة السابقة



الجدول مستمر من الصفحة السابقة...

المفتاح	الوظيفة
<b>PgDn</b>	الانتقال إلى الصفحة التالية
<b>Ctrl</b> - <b>PgUp</b>	القفز إلى أول عنصر في المنظومة
<b>Ctrl</b> - <b>PgDn</b>	القفز إلى آخر عنصر في المنظومة

وسيؤدي ضغط كافة المفاتيح الأخرى إلى أن يمرر التحكم إلى الوظيفة التي يحددها المستخدم مما سينفذ اختبارات شرطية وفق ذلك. تستدعي الوظيفة التي يحددها المستخدم أيضاً عندما تكون الوظيفة ( ) ACHOICE عاطلاً عن العمل (أي لا توجد مفاتيح للمعالجة).

تمرر الوظيفة ( ) ACHOICE تلقائياً ثلاثة متغيرات " للوظيفة المعرفة من قبل المستخدم الذي أعدناه: (أ) الطور الحالي ، (ب) العنصر الحالي في المصفوفة ، (ج) موضع الصف المناسب ضمن النافذة . وأول ما يجب أن تقوم به وظيفتنا هو أن تفحص قيمة الوظيفة ( ) LastKey.

أما الأطوار الممكنة للوظيفة ( ) ACHOICE (الثوابت الظاهرية الموجودة في ملف الترويسة CH.ACHOICE:

الطور	الثابت الظاهري	التوضيح
0	AC_IDLE	غير مستخدم (عديم الفائدة)
1	AC_HITTOP	محاولة نقل أول بند سابق
2	AC_HITBOTTOM	محاولة نقل آخر بند سابق
3	AC_EXCEPT	استثناء ضغط المفتاح
4	AC_NOITEM	لا توجد بنود منتقاة



يجب أن ترجع الوظيفة المعرفة من قبل المستخدم إحدى القيم إلى الوظيفة

:ACHOICE()

القيمة	الثابت الظاهري	التوضيح
0	AC_ABORT	الخروج بدون اختيار
1	AC_SELECT	الاختيار والخروج
2	AC_CONT	الاستمرار ( ) ACHOICE
3	AC_GOTO	الانتقال إلى البند التالي الذي يشبه حرفه الأول المفتاح الأخير

المثال البسيط التالي يربط وظيفة معرفة من قبل المستخدم بوظيفة ( ) ACHOICE. وستفحص وظيفتنا ( ) MyFunc البرنامج بحثاً عن مفاتيح "الخروج Esc" و "Enter" و "مسطرة المسافات Spacebar". لاحظ أن القيم الراجعة تختلف من مفتاح لآخر.

```
#include "inkey.ch"
#include "achoice.ch"

function main
local marray := {'one', 'tow', 'three'}, ele
cls
ele := achoice(11, 38, 13, 42, marray, .t., 'MyFunc')
return nil

function myfunc(status, curr_elem, curr_row)
local key := lastkey( )
if key == K_ESC
return AC_ABORT
elseif key == K_ENTER
return AC_SELECT
elseif key == 32
alert( "you pressed spacebar! Boy am I smart!" )
else
alert( "I don't understand that key!" )
endif
return AC_CONT
```

### ملاحظة

إن الوظائف التي يعرفها المستخدم لا يمكن أن تعلن من النوع الساكن STATIC ، وإلا فستكون "غير مرئية" للوظيفة ( ) ACHOICE (وبالتالي لا فائدة منها).



## كتابة الوظائف المعرفة من قبل المستخدم

### الوظائف مقابل الإجراءات Functions Vs. Procedures

إن الفارق الوحيد بين الوظائف والإجراءات هو أن الوظائف ترجع قيمة. فإن لم نكن نريد إرجاع قيمة من وحدة برنامج معينة ، فيمكننا أن نجعلها إجراءً. ويمكننا أيضاً كتابتها كوظيفة ترجع القيمة صفر NIL. وباختصار فإن كتابة الوظائف أو الإجراءات يعود إلى مايفضله المبرمج.

### وحدات البرامج Modularity

إن حجر الأساس لبرنامج كليبر 5.2 هو وحدات البرامج. ويجب ألا نكتب برنامجنا ضمن وظيفة واحدة معقدة ، بل يجب أن نقسمها في وحدات برامج أصغر. وكلما كان حجم الوظيفة صغيراً كلما كانت عملية الصيانة أسهل.

### التنظيم والتنسيق Housekeeping

يوفر برنامج كليبر 5.2 عدداً من الوظائف التي تسهل عملية تنظيم وتنسيق وحدة البرنامج ، وهي الوظائف التالية : "ضبط الألوان" ( setcolor ) "الصف" ( Row ) و "العمود" ( Col ) و "ضبط موضع المؤشر" ( setpos ) ، و "حفظ الشاشة" ( savescreen ) ، و "إعادة ضبط الشاشة" ( Resetscreen ) ، و "الضبط" SET ، و "ضبط المؤشر" ( Setcursor ) و "ضبط المفاتيح" ( Setkey ) ، و "ضبط الوميض" ( Setblink ).

## وظيفة ضبط الألوان ( SETCOLOR )

تمكننا هذه الوظيفة من فحص ، وتغيير الضبط الحالي للألوان. وهو ضروري لإعادة ضبط الألوان بعد تغييرها. عندما نستدعي وظيفة ضبط الألوان ( SETCOLOR ) ، فإنه يعيد الضبط الحالي للألوان. وإذا قمنا بتمرير سلسلة الألوان إليه كمتغير ، فإنه سيغير أيضاً ضبط الألوان حسب هذا المتغير. ويبين المثال التالي ذلك:

```
oldcolor = setcolor( 'w/r' )      // change color to white on red
```

```
setcolor(oldcolor)                // reset previous color
```

## وظيفة اختيار الألوان ( COLORSELECT )

إذا احتجنا لضبط الألوان الحالية (أي السلسلة الراجعة من قبل وظيفة ضبط الألوان ( Setcolor ) لتحديد أحد مجموعات الألوان ، فيجب علينا استخدام وظيفة اختيار الألوان ( COLORSELECT ). كما تمكننا هذه الوظيفة من تشغيل إحدى مجموعات الألوان الخمس دون تغيير قيمة وظيفة ضبط الألوان ( SETCOLOR ). ولتسهيل العملية يمكننا استخدام الثوابت الظاهرية الموجودة في ملف الترويسة COLOR.CH المتوفر مع رزمة كليبر 5.2. وهي كما يلي:

الثوابت الظاهرية	القيم
CLR_STANDARD	0
CLR_ENHANCED	1
CLR_BORDER	2
CLR_BACKGROUND	3
CLR_UNSELECTED	4

ويبين المثال التالي من البرنامج كيفية استخدام الوظيفة ( COLORSELECT ):

```
#include "color.ch"

function main
setcolor( 'w/r, +w/b,,,+gr/g')
```

```
? colorselect(CLR_ENHANCED)
? "displays bright white on blue"
? colorselect(CLR_UNSELECTED)
? "displays yellow on green"
? colorselect(CLR_STANDARD)
? "displays white on red"
return nil
```

## وظائف كل من ROW( ) / COL( ) / SETPOS( )

يحتمل جداً أن تغير كثيراً من الوظائف مكان المؤشر. فراجع كل من وظيفتي COL( ) , ROW( ) المؤشر إلى السطر والعمود الحاليين ، حسب الترتيب المبين ، فيجب الإنتباه إلى استدعائهما قبل نقل المؤشر أو تحريكه إلى مكان آخر.

استخدم الوظيفة SETPOS( ) إذا أردت إعادة تجهيز المؤشر ، وتقبل هذه الوظيفة متغيرين وهما السطر والعمود <Column> و <Row> وتنقل المؤشر إلى ذلك المكان المحدد وتبين الشيفرة التالية كيفية استخدام الوظائف الثلاثة لحفظ مكان المؤشر وإعادةه على الشاشة كما كان قبل التغيير:

```
oldrow := row( )
oldcol := col( )
```

## وظيفةتا SAVESCREEN( ) و RESETSCREEN( )

تعتبر هاتان الوظيفتان عمليتين جداً إذ تمكنك من حفظ محتويات الشاشة بأكملها أو أجزاء منها ، أو إعادةتها إلى ما كانت عليه قبل التغيير. وأما التركيب اللغوي لوظيفة SAVESCREEN( ) فهو كما يلي:

```
SAVESCREEN( [<top>, <left>, <bottom>, <right> ] )
```

ويحدد كل من الاتجاهات الأربعة فوق ، يسار ، أسفل ، يمين المنطقة المراد حفظها من الشاشة. أما إذا لم تستخدم تحديد هذه الجهات فسيعمل البرنامج على حفظ محتويات

الشاشة بأكملها بغض النظر عن أمكنتها. مثل: ( MAXCOL( , MAXROW( , 0, 0 ).  
كما أن الوظيفة ( Savescreen( ترجع السلسلة الحرفية التي يجب الاحتفاظ بها في  
متغير ما أو في عنصر مصفوفة إلى أن يأتي الوقت الذي ترغب إعادتها.

والتركيب اللغوي لهذه الوظيفة كما يلي :

RESTSCREEN( <top> , <left> , <bottom> , <right> , <buffer> )

حيث تحدد كل من الاتجاهات الأربعة المنطقة التي يجب إعادتها كما كانت في وضعها  
السابق قبل التغيير.

#### ملاحظة

إذا كان حجم الشاشة التي سبق حفظها يختلف عن الحجم المحدد حالياً باستخدام هذه  
الوظيفة فيجب أن تتوقع حدوث اختلافات في مظهر الشاشة وشكلها ، أما إذا رأيت  
"بعض الوجوه المضحكة ، إلى جانب عدد من رموز آسكي" أمامك على الشاشة بعد  
إعادة الشاشة إلى وضعها السابق قبل التغيير فيمكن الظن بأن المشكلة هي من هذا  
النوع.

ولنبين فيما يلي مثلاً عن كيفية استخدام هاتين الوظيفتين:

```
local b := savescreen( )
cls
inkey( 0 )
restscreen ( 0, 0, maxrow( ) , maxcol( ) , b)
```



## وظيفة التجهيز ( SET )

تقبل هذه الوظيفة متغيرين ، الأول هو الثابت الظاهري الذي يمثل التجهيز الذي ترغب التساؤل عنه أو تغييره. ويحتوي ملف SET.CH هذا النوع من الثوابت ، والتي نبينها أدناه. وستلاحظ أن مصطلحات التسمية سهلة التذكر. وأما المتغير الخياري الثاني فهو قيمة التغير الذي تريد إجراؤه. (لاحظ بأنه لا يجب عليك تضمين ملف الترويسة SET.CH ، وذلك لأنه يتم استدعاء هذه القيم آلياً).

وترجع وظيفة التجهيز ( SET ) حالة الشاشة إلى الوضعية المطلوبة ، ويمكن حفظها في متغير ما ، واسترجاعها في أي وقت من الأوقات تريد.

### جدول - الثوابت الظاهرية

نوع بيانات المتغير الثاني	الثوابت الظاهرية الموجودة في SET.CH
Logical	_SET_EXACT
Logical	_SET_FIXED
Numeric	_SET_DECIMALS
character	_SET_DATEFORMAT
Numeric	_SET_EPOCH
character	_SET_PATH
character	_SET_DEFAULT
Logical	_SET_EXCLUSIVE
Logical	_SET_SOFTSEEK
Logical	_SET_UNIQUE
Logical	_SET_DELETED
Logical	_SET_CANCEL
Logical	_SET_DEBUG
character	_SET_COLOR
Numeric	_SET_CURSOR
Logical	_SET_CONSOLE
Logical	_SET_ALTERNATE
character	_SET_ALTFILE*
character	_SET_DEVICE
character	_SET_EXTRAFILE*
Logical	_SET_PRINTER
character	_SET_PRINTFILE*
Numeric	_SET_MARGIN
Logical	_SET_BELL

الجدول مستمر من الصفحة السابقة

نوع بيانات المتغير الثاني	الثوابت الظاهرية الموجودة في SET.CH
Logical	_SET_CONFIRM
Logical	_SET_ESCAPE
Logical	_SET_INSERT
Logical	_SET_EXIT
Logical	_SET_INTENSITY
Logical	_SET_SCOREBOARD
Logical	_SET__DELIMITERS
character	_SET_DELIMCHARS
Logical	_SET_WRAP
Numeric	_SET_MESSAGE
Logical	_SET_MCENTER
Logical	_SET_SCROLLBREAK

كما يمكنك أيضاً تمرير قيمة منطقية كمتغير ثالث للتجهيزات الشاملة. وهو: الطابعة PRINTER (كعادة توجيه إخراجها إلى ملف نص): أو "بدل" ALTERNATE ، أو "إضافي" EXTRA. وأما الثوابت الظاهرية لهذه التجهيزات فهي \_SET\_PRINTFILE و \_SET\_ALTFILE و \_SET\_EXTRAFILE على التوالي. وأما إذا تمرير القيمة المنطقية مثل "حقيقي" (T.) والذي يعني : إذا وجدت الملف "الهدف" فيجب إلحاقه بالملف المطلوب بدلاً من استبداله ، والكتابة فوقه. وإذا استخدمت العبارة ADDITIVE مع أي من وظيفتي SETPRINTR TO أو SETALTERNATE TO فإن المتغير الحقيقي سيتم تضمينه في هذا الأمر بشكل آلي.

### وظيفة ( ) SETCURSOR (تجهيز المؤشر)

تعمل هذه الوظيفة على غرار مثيلتها ( ) SETCOLOR إلا أنها تتعامل مع حجم المؤشر بذاته. فإذا لم متغيراً محدداً ، فإنها ترجع شكل المؤشر على ما كان عليه في السابق ، قبل التغيير ، والذي يمكن أن يكون واحداً من الأشكال الخمسة التالية:

وصف شكل المؤشر	الثابت الظاهري المرتبط به والموجود في ملف الترويسة SETCURS.CH
بدون مؤشر	SC_NONE
عادي (تحت خط)	SC_NORMAL
وضع الإزاحة (كتلة نصف سفلية)	SC_INSERT
كتلة كاملة	SC_SPECIAL1
كتلة نصف علوية	SC_SPECIAL2

فإذا مررت متغيراً رقمياً فإن هذه الوظيفة ستغير شكل المؤشر إلى الشكل المطلوب الذي حدد رقمه.

ويجب تضمين العبارة `#include "setcurs.ch"` لاستخدام الثوابت الظاهرية المبينة أعلاه في مكان ما في البرنامج (المصدر) قبل الإشارة إليها واستخدامها.

## وظيفة تجهيز مفتاح ( ) SETKEY

تشبه هذه الوظيفة مثيلتها ( ) SET وتتيح للمستخدم تغيير استخدام مفتاح ما كان قد عُرفَ على أنه مفتاح مباشر ( Hot Key ). وتقبل هذه الوظيفة متغيرين اثنين هما:

- المتغير الأول هو القيمة الرقمية للمفتاح INKEY المراد اختباره. وبدلاً من الإشارة إلى الأرقام مباشرة ، فإننا نقترح استخدام الثوابت الظاهرية التي يحتوي عليها ملف الترويسة INKEY.CH وسنبين هذه الطريقة أدناه.
- المتغير الاختياري الثاني يمكن أن يكون واحداً من اثنين: (أ) كتلة برنامج code block يمكن عندئذ ربطها بذلك المفتاح ويتم تقييمها عند الضغط عليه في برنامج كليبر. أو (ب) صفر NIL والذي يوقف عمل المفتاح المباشر بشكل فعال. أما المبرمجون الذين لا يزالون راغبين باستخدام برمجة الكتل code blocks ، فيمكنهم

استخدام أمر SETKEY لضبط حالة المفتاح ووضعته الفعلية. ويبين المثال التالي الطريقتين المذكورتين.

ترجع الوظيفة ( ) SETKEY القيمة NIL (صفر) إذا لم يكن المفتاح مفتاحاً مباشراً ، وأما إذا كان المفتاح قد حدد ليستخدم على أنه مفتاح مباشر فإنها ترجع كتلة البرمجة المرتبطة به. ويمكنك عندئذ إعادة تعيين كتلة البرمجة من جديد إليه عندما تنتهي من التعامل معه.

## مثال:

```
#include "setcurs.ch"    // necessary for cursor constants
#include "inkey.ch"      // necessary for keypress constants

function main
// statements
myfunc( )
// statements
return nil

function myfunc
local oldcursor := setcursor(SC_NONE)      // turn off cursor
local olddel    := set(_SET_DELETED, .T.)  // don't show deleted recs
local oldscrn   := savescreen( )
local oldcolor  := setcolor( )
// note: using SET KEY command to set F1 status below
local oldf1     := setkey(K_F1)
set key K_F1 to subhelp
//
// body of function
//
setcursor(oldcursor)           // restore previous cursor status
set(_SET_DELETED, olddel)      // restore previous DELETED status
setkey(K_F1, oldf1)            // restore previous F1 status
setcolor(oldcolor)             // restore previous color setting
restscreen(0, 0, maxrow( ), maxcol( ), oldscrn)
return nil
```

## وظيفة ( ) SETBLINK

يجب أن نتطرق للوظيفة ( ) SETBLINK طالما أننا مازلنا نتحدث عن هذا الموضوع. فكثير من المستخدمين يحبون الألوان الساطعة كخلفية للبرامج التي يتعاملون مع شاشاتها (وخاصة اللون الأصفر). ويمكنك الاختيار ما بين الواجهة التي تضيء وتعتم قليلاً أو الخلفية الساطعة اللون. فإذا وضعت "الإضاءة والتعتيم" في وضعية الإيقاف باستخدام الوظيفة ( ) SETBLINK فستكون خلفية الشاشة ساطعة اللون.

وقد تم بناء هذه الوظيفة بشكل مماثل لوظيفة ضبط اللون ( ) SETCOLOR أي أنها تعود دائماً إلى وضعية التجهيز الحالية ، وهي : "بت الإضاءة والتعتيم" وهي تقبل متغيراً منطقياً بشكل اختياري يضبط وضعية هذه "البت". كما أن تمرير القيمة المنطقية F. فسيشغل هذا لون الخلفية الساطعة ، بينما يلاحظ أن تمرير القيمة المنطقية T. سيشغل "الإضاءة والتعتيم". في المثال التالي يتم التبادل بين الإضاءة والتعتيم للحصول على خلفية ذات لون أصفر:

```
function main
local oldblink := setblink(f.)
@ 0,0,maxrow( ),maxcol( ) box "*****" color '*n/gr'
inkey(0)
setblink(oldblink)
return nil
```

## استقلالية طور الفيديو

يمكن برمجة برامجك بمنتهى السهولة بحيث يمكنك التأقلم تلقائياً مع أية وضعية من وضعيات الفيديو التي يطلبها المستخدم. بل ويمكن أيضاً تسهيل استخدام ذلك على المستخدم ذاته للاختيار ما بين العرض على ٢٥ سطراً أو ٤٣ سطراً أو ٥٠ سطراً على الشاشة من داخل البرنامج ذاته.



أما الوظائف الثلاثة التي تمكنك من القيام بهذا فهي ( ) SETMODE و ( ) MAXROW و ( ) MAXCOL ، وإليك كيفية التعامل مع كل منها:

## وظيفة ( ) SETMODE (ضبط الوضعية)

تمكنك هذه الوظيفة من تغيير وضعية عرض الفيديو ، وتقبل متغيرين رقميين هما السطر <row> والعمود <column> وتحاول التغيير بالانتقال إلى الوضعية المناسبة. ويمكن التجاوز عن أي من هذين المتغيرين إذا لم ترغب تغيير تلك الخاصية (إذا أردت تغيير عدد الأسطر فقط ، استخدم الطريقة التالية (50) SETMODE ) وتعيد هذه الوظيفة قيمة منطقية True (حقيقي) إذا تم تغيير الوضعية بنجاح ، أو False (غير حقيقي) إذا أخفقت عملية التغيير. وإن النجاح والإخفاق يعتمدان على الأجهزة والكروت المستخدمة.

## وظيفتا ( ) MAXROW / ( ) MAXCOL

وظيفة الحد الأقصى للأسطر والحد الأقصى للأعمدة ( ) MAXCOL ، ( ) MAXROW تعيد هاتان الوظيفتان وضعيتي الحد الأقصى للأسطر أو للأعمدة والتي يمكن عرضها على الشاشة وقيم الإرجاع النموذجية لهاتين الوظيفتين هما ٢٤ و ٧٩ إذ أن معظم الأعمال التي يؤديها مستخدم الكمبيوتر إما أن تكون بوضعية النصوص القياسية وهي ٢٥ سطراً × ٨٠ عموداً ، إلا أن وظيفة ( ) SETMODE تسهل استخدام وضعيات عرض مختلفة في البرامج المستخدمة ، لذلك ننصح باستخدام كل من ( ) MAXROW و ( ) MAXCOL بحيث يمكن التخطيط لاستخدام البرامج بشكل مناسب ، مثال:

```
oldscrn = savescreen(0, 0, maxrow( ), maxcol( ))
restscreen(0, 0, maxrow( ), maxcol( ), oldscrn)
```



وعند توسيط نص ما على الشاشة استخدم الوظيفة MAXCOL+1 كعرض للشاشة بدلاً من ٨٠ عموداً ، إذ قد لا تكون دائماً في وضعية ٨٠ عموداً.

وتبين شيفرة برنامج النافذة التالي مثلاً على كل من هذه الوظائف الثلاث عملياً. وقد تم رسم شاشة عنوان تتضمن إطاراً وضع في وسط الشاشة. ويمكنك أن تضغط على مفتاح [F1] لتشغيل وضعية الفيديو أو إيقافها. ويجب أن تلاحظ أثناء قيامك بذلك أن الإطار يبقى في وسط الشاشة ويحافظ على موقعه هناك. كما يجب ملاحظة السلسلة الأقحوانية لكل من وظيفتي SETMODE(50) و SETMODE(43) ، وقد تم عمل هذا بشكل مقصود بحيث يحاول البرنامج أولاً استخدام وضعية ٥٠ سطراً وإذا أخفق في هذا يحاول وضعية ٤٣ سطراً. ويعتبر هذا أمر ضرورياً لأن كثيراً من شاشات VGA ومهايئاتها تستطيع التعامل مع الوضعتين المشار إليهما.

```
1  #include "box.ch"
2  #include "inkey.ch"
3
4  function main
5  local oldrows := maxrow( )
6  local oldcol  := maxcol ( )
7  videodemo( )
8  // reset video mode if it was changed
9  if maxrow( ) <> oldrows .or. maxcol( ) <> oldcols
10     // change color/clear screen prior to setmode( ) to avoid "flash"
11     setcolor('w/n')
12     scroll( )
13     setmode(oldrows + 1, oldcols + 1)
14 endif
15 return nil
16
17
18 /*
19 Function: VideoDemo( )
20 Purpose: Stub function to demonstrate toggling video mode
21 */
22 function videodemo
23 local key
24 titlescreen( )
25 do while ( key := inkey(0) <> K_ESC
26     if key == K_F1
27         togglemode( )
```

```
28     endif
29 enddo
30 return nil
31
32
33 /*
34 Function: changeMode( )
35 Purpose: change video mode and redraw title screen
36 */
37 function togglemode
38 local success
39 if maxrow( ) > 25
40     success := setmode(25)
41 else
42     success := (setmode(50) .or. setmode(43) )
43 endif
44 if success
45     titlescreen( )
46 endif
47 return nil
48
49
50 #define BACK_COLOR '+w/b'
51 #define INFO_COLOR 'n/bg'
52
53 /*
54 Function: TitleScreen( )
55 Purpose: Draw title screen
56 */
57 function titlescreen
58 local midrow := int( maxrow( ) / 2 )
59 local midcol := int( maxcol( ) / 2 )
60 @ 0, 0, maxrow( ), maxcol( ) box replicate(chr(197), 9) color BACK_COLOR
61 @ midrow - 2, midcol - 14, midrow + 2, midcol + 14 ;
62     box B_SINGLE + ' ' color INFO_COLOR
63 @ midrow-1, midcol-12 say "Video mode demonstration" color INFO_COLOR
64 @ midrow, midcol-10 say "Now viewing " + ltrim(str(maxrow( )+1)) + ;
65     " lines" color INFO_COLOR
66 @ midrow+1, midcol - 12 say "F1 = toggle  ESC = quit" color INFO_COLOR
67 return nil
```

## الوظائف الساكنة Static Functions

إذا قمنا بالإعلان عن وظيفة ما ، أو إجراء ما بأنه ساكن Static فإن هذا يحد من رؤيته على وظائف أو إجراءات أخرى فقط ضمن ملف البرنامج ذاته PRG.. ولاشك أن هذه الطريقة تحد كثيراً من التعارض بين أسماء الوظائف المختلفة.

ولعل أحد الأمثلة البارزة على تضارب الأسماء ، الوظيفة المشار إليها أعلاه وهي الوظيفة CENTER( ) إذ أن كل مبرمج له طريقته الخاصة في عرض التركيب اللغوي المستخدم في هذه الوظيفة بحيث يتسبب هذا في تضارب وفوضى لا حد لها. إلا أنه يمكنك إخفاء أسماء وظائفك الخاصة عن المبرمجين الآخرين العاملين معك على البرنامج ذاته ، بإعلان هذه الوظائف على أنها ساكنة STATIC. ويمكنك الاطلاع على البرنامج التالي:

```
/* MAIN.PRG */
function main
whatever( )
center(16, "Ouch!") // run-time error
return nil

* eof main.prg

/*-----*/

/* FUNCS.PRG */
function whatever
center(17, "This is a test")
return nil

static function center(row, msg)
@ row, int ((maxcol( ) + 1 - len(msg)) / 2) say msg
return nil

* eof funcs.prg
```

ولن تكون وظيفة التوسيط CENTER( ) مرئية بالنسبة للوظائف والإجراءات الأخرى ضمن ملف FUNCS.PRG. وعند استدعاء الوظيفة ( ) WHATEVER من ملف MAIN.PRG . فلن تواجه أية مشكلة لدى استدعاء وظيفة التوسيط CENTER( )

لأنهما في ملف البرنامج ذاته ، إلا أنك إذا حاولت استدعاء الوظيفة ( ) CENTER مباشرة من ملف MAIN.PRG فسيتعطل تنفيذ البرنامج.

وهكذا ، فإنك باستخدام الوظائف الساكنة STATIC يمكن أن يكون لديك عدد من الوظائف تحمل الاسم ذاته خلال برنامجك طالما أنها جميعها موجودة في ملفات برامج متفرقة.

فإذا كان عمل المبرمج مع فريق من المبرمجين فإنه ، دون شك ، سيحب هذه الميزة تماماً وليس لأنها تحد من تضارب الأسماء وتعارضها فقط ، بل لأنها أيضاً تسهل عملية كتابة الوثائق المتعلقة بالبرامج أيضاً. ولنفرض أنك تعمل على إعداد وحدة برمجية تشتمل على العديد من الوظائف على النحو التالي:

```
function entry( )
*
return nil

static function func1( )
*
return nil

static function func2( )
*
return nil

etceters
```

فعندما يحين الوقت لتوزيع وحدتك البرمجية إلى المبرمجين الآخرين في المجموعة ذاتها فلن تحتاج إلى كتابة توثيق سوى المتغيرات اللازمة لوظيفة الإدخال فقط ( ) Entry. ولن يكونوا بحاجة إلى معرفة الوظائف الساكنة والتي لا تستخدم إلا في ملف PRG. الخاص بك.

### قاعدة عامة

إذا أردت معرفة ما إذا كانت وظيفة ما يستحسن أن تعلن أنها "ساكنة" اسأل نفسك مايلي: "هل أحتاج لاستدعاء هذه الوظيفة من خارج ملف هذا البرنامج؟". فإذا كان الجواب بالنفي فيستحسن أن تجعلها ساكنة `STATIC`.







## شاشات إدخال البيانات

إن هذه الشاشات من أهم أجزاء أي برنامج من البرامج التي يتم إعدادها باستخدام برنامج كليبر. ويجب أن تتألف هذه الشاشات من الأجزاء التالية ، وهي:

١- عرض النص الساكن (وعادة يتم هذا باستخدام أمري: @..SAY و @..BOX ).

٢- إنشاء متغيرات يمكنها احتواء نسخ من حقول قاعدة البيانات.

٣- عرض متغيرات لإدخال البيانات باستخدام @..GET .

٤- إذا لم يخرج المستخدم من شاشة الإدخال ، وكان يضيف سجلاً فيمكن استخدام APPEND BLANK (أضف سجلاً فارغاً).

٥- إذا لم يخرج المستخدم من شاشة الإدخال ، وكتب القيم في المتغيرات في حقل قاعدة البيانات ( مفترضاً أن السجل أو الملف مقفل أو أن الملف يستخدم بشكل خاص exclusively).

ونبين فيما يلي مثلاً على شاشة إدخال البيانات ، وهي شاشة ذات أغراض متعددة يمكنك إضافة سجلات ، أو تعديلها أو ستعراضها. وتقبل الوظيفة ( AddEdit متغير الوضعية ، والذي قد يكون إما "A" أو "E" أو "V" وبمعنى أوضح إضافة ، تعديل ، عرض.

```
#include "inkey.ch"
#include "setcurs.ch"
```

```
function addedit(mode)
local lOldinsert := setkey( K_INS, { || setcursor( ;
    if (readinsert( .not. readInsert( )), SC_NORMAL, SC_SPECIALI) ) } )
local getlist := { }
local lOldscors := set(_SETSCOREBOARD, .F. )
local nOldrow := row( )
local nOldcol := col( )
local cOldcolor := setcolor( )
local cOldscrn := savescreen( )
```

```

local nOldcursor := setcursor( )
local lContinue
local name
local title
local date
local read
use articles shared
cls
// ---- static text
setcolor('+w/b,+w/n,,,+W/B')
@ 9, 33 say "Name"
@ 10, 32 say "Title"
@ 11, 32 say "Date"
@ 12, 33 say "Read"
// ---- use phantom record to initialize variables
if mode == 'A'
    marker := recno( )
    go 0
endif
// ---- initialize field duplicates
name := articles->name
title := articles->title
date := articles->date
read := articles->read
// ---- date entry
@ 9, 39 get name picture '@!' VALID ! empty(name)
@ 10, 39 get title picture '@!'
@ 11, 39 get date
@ 12, 39 get read picture 'Y'
if mode
    setcursor(if(readInsert( ), SC_SPECIAL1, SC_NORMAL) )
    read
    if lastkey( ) <> K_ESC
        // ---- add blank record or lock the record
        if mode == 'A'
            append blank
            lContinue := .not. netem( )
        else
            lContinue := rlock( )
        endif
    endif
    if lContinue
        // ---- copy values back to field
        articles->name := name
        articles->title := title
        articles->date := date
        articles->read := read
    endif
    // ---- if in add mode, reset record pointer

```

```

elseif mode == 'A'
    go marker
endif
else
    getlist := {} // clears pending GETs above
    inkey(0)
endif
// ---- restores environment
setcursor*nOldcursor)
setpos(nOldrow, nOldcol)
setcolor(cOldcolor)
restscreen(0, 0, maxrow( ), maxcol( ), cOldscrn)
setkey(K_INS, IOldinsert)
set(_SET_SCOREBOARD, IOLDSCORE)
return nil

```

وتجدر الإشارة إلى أنه تم إيقاف عمل SCOREBOARD في أعلى الوظيفة ، وهي منطقة في الشاشة على السطر (صفر) والعمود (60) تستخدم لعرض وضعية الإدخال. إلا أنه بسبب حجز ( لوحة النتيجة ) Scoreboard لهذه المنطقة ، فقد يسبب هذا اضطراباً كبيراً بسبب البقعة غير الظاهرة لكل من المبرمجين والمستخدمين على حد سواء. وعلاوة على ذلك ، فليس هذا المكان مفضلاً لعرض هذه النتيجة في أي حال من الأحوال ، فيرجى الانتباه إلى هذا الموضوع أثناء إعداد البرامج.

لذلك ، فبدلاً من استخدام "لوحة النتيجة" SCOREBOARD المشار إليها ، يستحسن استخدام الوظيفة ( ) SETKEX لضبط تشغيل عمل مفتاح وظيفة طور الإقحام insert أو الكتابة على الموجود overstrike ، أو إيقافه وكذلك حجم المؤشر. وهذا ما تستخدمه كثير من برامج معالجة النصوص وهو التعامل مع طور الإقحام.





## معالجة السلاسل والمذكرات

### وظائف السلاسل

يقدم برنامج كليبر 5.2 العديد من وظائف التنسيق الخاصة بمعالجة السلاسل الحرفية. وتعتبر هذه الوظائف مفيدة بشكل خاص فيما يتعلق بإخراجات كل من الشاشة والطابعة ، كما يمكن الإفادة منها لدى تأسيس حقول قواعد البيانات أو المتغيرات.

#### الوظائف ALLTRIM( ) و LTRIM( ) و RTRIM( )

تمكنا هذه الوظائف من حذف الفراغات السابقة أو التالية ( أو كليهما معا) من السلاسل الحرفية. وتقبل كل منها السلسلة الحرفية كمتغير. فتعمل الوظيفة LTRIM( ) على حذف الفراغات السابقة ، بينما تعمل وظيفة RTRIM( ) على حذف المسافات اللاحقة ، وتعمل الوظيفة ALLTRIM( ) على حذف كل المسافات الفارغة قبل السلسلة الحرفية وبعدها.

وتعتبر هذه الوظائف مفيدة لتنسيق البيانات الحرفية وخاصة عندما تريد الربط بين بيانات حرفية موجودة في حقليْن مختلفين على النحو التالي:

```
? rtrim( customer->first ) + ' ' + customer->last
```

## الوظائف PADL( ) و PADC( ) و PADR( )

تعمل هذه الوظائف بعكس سابقاتها ، أي أنها تضيف بعض المسافات الفارغة قبل السلسلة الحرفية أو بعدها ، أو من الجانبين: قبلها وبعدها. والتركيب اللغوي لها جميعها هو واحد:

PAD( < data >, < nLength > [ , < cFill > ] )

وبما أن المتغير < data > إما أن يكون سلسلة حرفية ، أو تاريخاً أو تعبيراً رقمياً يجب العمل عليه.

والمتغير < nLength > هو الطول المطلوب.

وأما المتغير الاختياري < cFill > فيمثل: الحرف الذي ستملأ به فراغات السلسلة الحرفية المطلوبة. فإذا لم تحدد المتغير < cFill > ، فسيستخدم الفراغ (الأسكي 32).

وترجع هذه الوظائف الثلاث سلسلة حرفية محشاة Padded وهي ذات طول < nLength > تحتوي على البيانات المحدد في المتغير < data >. وتقوم الوظيفة PADL( ) بحشو السلسلة بمسافات فارغة ، وأما الوظيفة PADR( ) فهي تسحب المسافات ، والوظيفة PADC( ) توسط البيانات الموجودة في المتغير < data > وذلك بإضافة مسافة على كلا الجانبين من البيانات.

وتجدر الإشارة إلى أنه إذا كان المتغير < nLength > أقصر من بداية طول البيانات < data > فإن الوظيفة PAD( ) ستبتر الحقل < data > طبقاً لذلك.



### فكرة مفيدة

يمكن استخدام هذه الوظائف لمخرجات الشاشات ، حيث أن هذه الوظائف تضمن أن النص المكتوب سابقاً قد تلاشى عند كتابة النص الجديد.

وإليك بعض الأمثلة على عمل هذه الوظائف:

## وظيفة النسخ REPLICATE( )

تكرر هذه الوظيفة نسخ سلسلة حرفية ما بعدد المرات الذي تحدده لها. والتركيب اللغوي لها هو كما يلي:

REPLICATE ( < cString > , < nRepead > )

حيث يكون المتغير <cString> هو السلسلة الحرفية المراد تكرارها. وإذا حددت صفراً للمتغير الثاني <nRepeat> فستعيد الوظيفة REPLICATE( ) سلسلة فارغة تماماً.

وتجدر الإشارة إلى أن الحد الأقصى لطول هذه الوظيفة REPLICATE( ) هو ٦٥٥٣٥ حرفاً.

وإليك مثالين يبينان طريقة استخدام هذه الوظيفة:

padl ( "test" 10)	// returns	" test"
padr ( "test" 10)	// returns	"test "
padc ( "test" 10)	// returns	" test "
padl (5, 10, "**")	// returns	"*****5"
padl( date( ), 10)	// returns	"09/23/92 "

## الوظيفة SPACE ( )

تقبل هذه الوظيفة متغيراً رقمياً وترجع سلسلة رقمية تحتوي على ذلك العدد من المسافات. ويمكن أن تكون هذه السلسلة الرقمية بحد أقصى ٦٥٥٣٥ رمزاً. وتستخدم هذه الوظيفة لربط السلاسل أثناء الطباعة ، ويمكن استخدامها أيضاً لتأسيس متغيرات فارغة.

## الوظيفة SUBSTR ( )

تتمكنك هذه الوظيفة المفيدة من مشاهدة جزء من السلسلة فقط والتركيب اللغوي لهذه الوظيفة هو:

SUBSTR ( <cString> , <nStart> [ , <nLength> ] )

لا بد وأنك قد عرفت من النظر في هذه القاعدة اللغوية بأن السلسلة الحرفية <cString> هي المقصودة بالسؤال. والمتغير <nStart> هو الموقع المطلوب الانطلاق منه في السلسلة <cString>. أما المتغير الاختياري <nLength> فهو عدد الحروف المطلوبة. مع العلم أنك إذا أغفلت ذكر هذا المتغير الأخير ، فإن الوظيفة SUBSTR ( ) ستبحث من <nStart> إلى أن تصل إلى نهاية السلسلة.

وتجدر الإشارة أيضاً إلى أنه يمكن تحديد رقم سلبي لنقطة البداية <nStart> حيث يبدأ العداد في مثل تلك الحالة من النهاية بدلاً من البداية ، على النحو التالي:

```
name := "AAAA BBBB"
? substr(name, 1, 4) // "AAAA"
? substr(name, 6)   // "BBBB"
```

## وظيفة LEFT( ) و RIGHT( )

تشبه هاتان الوظيفتان الوظيفة SUBSTR( ) بأنهما يمكنانك من مشاهدة أجزاء من السلسلة الحرفية بدءاً من اليسار أو اليمين وإن التركيب اللغوي لهذه الوظائف هو متماثل تماماً.

LEFT | RIGHT( <cString>, <nCount> )

وبالطبع ، فإن <cString> هو سلسلة حرفية يمكن مشاهدتها ، وأما <nLength> فهو عدد الحروف التي يجب تضمينها فيها.

ويبين الجزء التالي من البرنامج هذه الوظائف:

```
name := "AAAA BBBB"
? left(name, 4)      // "AAAA"
? right(name, 4)     // "BBBB"
```

## وظيفة LOWER( ) و UPPER( )

تغير هاتان الوظيفتان السلاسل الحرفية من الحروف الصغيرة إلى الحروف الكبيرة – وذلك بالنسبة للغة الانجليزية ، حيث لا يوجد حروف كبيرة وصغيرة في اللغة العربية – وكلاهما تقبل السلسلة الحرفية كمتغير ، ثم ترجع السلسلة المقلوبة ( المحولة ). ويبين المثال التالي كيفية عمل هذه الوظيفة:

```
?upper(left(name, 1)) + lower(substr(name, 2))
```

### فكرة مفيدة

تذكر أن تستعمل الوظيفة UPPER( ) إذا أردت أن يكون مفتاح الفهرسة index key لديك حساساً لحالة الحروف الكبيرة case-insensitive.

## وظائف المذكرة ( ) MEMO

يقدم برنامج كليبر عدداً من الوظائف التي تتعامل مع حقول المذكرة ، ويمكن تطبيق هذه الوظائف جميعها على السلاسل الحرفية أيضاً.

### وظيفة تحرير المذكرة ( ) MEMOEDIT

تعتبر هذه الوظيفة أسهل الوظائف استخداماً وأهمها لكتابة المذكرات ، وهي تمكنك من كتابة المذكرة أو عرضها. والتركيب اللغوي لها هو كما يلي ، ونأمل ألا ترعجك كثرة المتغيرات المطلوب تجهيزها ، إذ أنك لن تستخدم سوى بعض منها.

```
MEMOEDIT( [<cString>], [<nTop>], [<nLeft>], [<nBottom>], [<nRight>], ;
          [<lEdit>], [<cFunction>], [<nLength>], [<nTab>], [<nTextrow>], ;
          [<nTextcolumn>], [<nWindwrow>], [<nWindwcolumn>] )
```

والمتغير <cString> هو سلسلة ، أو المذكرة التي تنسخ في الذاكرة المؤقتة لـ: MEMOEDIT( ) وإذا لم تحدد هذا المتغير فستبدأ الذاكرة المؤقتة للنص فارغة.

وأما المتغيرات أعلى وأسفل ويسار ويمين فهي تحدد منطقة الشاشة التي ستنسخ فيها وظيفة ( ) MEMOEDIT.

وأما المتغير <lEdit> فهو قيمة منطقية تقرر ما إذا كان المستخدم يمكنه فعلاً تحرير المذكرة أم لا. فإذا كان هذا المتغير "حقيقياً" فيتم تشغيل قدرة التحرير ، وقد تم افتراض القدرة على تشغيل إمكانية التحرير مباشرة من قبل النظام.

أما المتغير <cFunction> فهو اسم الوظيفة المعرفة من قبل المستخدم والتي ستستدعى عند الضغط على أي مفتاح ضمن ( ) MEMOEDIT. فإذا أردت عرض المذكرة ثم

الخروج منها مباشرة يمكن تمرير القيمة المنطقية "غير حقيقي" F. كمتغير. ويعتبر هذا الأمر مفيداً إذا أردت أستعراض المذكرة فقط ، ثم الانتقال للقيام بعمل آخر.

أما المتغير <cLength> فهو يحدد طول كل سطر في نافذة ( ) MEMOEDIT. فإذا كان هذا أعرض من طول سطر النافذة فستدور النافذة لدى أنتقال المؤشر إلى مابعد الطرف الأيمن لها. وإذا لم تحدد هذا المتغير فسيكون المتغير المفترض المتغير الأيمن <nRight> أقل من المتغير الأيسر <nLeft>.

وأما المتغير <nTab> فهو يحدد عدد الفراغات المستخدمة لمسافة الجدولة Tab. وأما القيمة المفترضة فهي ٤ مسافات.

وأما المتغيران <nTextrow> و <nTextColumn> فهما يحددان موقع المؤشر ضمن الذاكرة المؤقتة لدى الدخول في ( ) MEMOEDIT. وتجدر الإشارة إلى أن السطور تبدأ بالرقم ١ ، والأعمدة تبدأ من الرقم صفر. وإذا لم يحدد هذان المتغيران فسيوضع المؤشر في السطر الأول العمود صفر ، أي الزاوية العليا اليسرى للشاشة.

أما المتغيران <nWindowrow> و <nWindowcolumn> فهما يحددان موقع المؤشر بالنسبة لموقع النافذة عند الدخول لأول مرة في ( ) MEMOEDIT. وزيادة في الإرباك فكل من السطور والأعمدة تبدأ من الصفر. فإذا لم تحدد ههما ، فسيكون موقع المؤشر على السطر صفر والعمود صفر ، أي الركن الأيسر العلوي.

يمكنك الخروج من الوظيفة ( ) MEMOEDIT باستخدام أحد المفاتيح التالية: [Ctrl] - [W] أو [Esc]. باستخدام الطريقة الأولى سيتم الاحتفاظ بما تم كتابته في المذكرة ، وأما الطريقة الثانية فستحتفظ بالسلسلة الأصلية التي كانت فيها من قبل فقط.

وتعتبر واجهة الوظيفة ( ) MEMOEDIT مشابهة تماماً لبرنامج ورد ستار. وتعتبر المفاتيح التالية نشيطة في حالة استخدام الوظيفة ( ) MEMOEDIT:

## جدول المفاتيح النشطة داخل الوظيفة MEMOEDIT()

المفتاح	الوظيفة
↑ or Ctrl-E	الانتقال إلى الأعلى سطر واحد
↓ or Ctrl-X	الانتقال إلى الأسفل سطر واحد
← or Ctrl-S	الانتقال إلى اليسار حرف واحد
→ or Ctrl-D	الانتقال إلى اليمين حرف واحد
Ctrl-← or Ctrl-A	الانتقال إلى اليسار كلمة واحدة
Ctrl-→ or Ctrl-F	الانتقال إلى اليمين كلمة واحدة
Home	الانتقال إلى بداية السطر الحالي
End	الانتقال إلى نهاية السطر الحالي
Ctrl-Home	الانتقال إلى بداية النافذة الحالية
Ctrl-End	الانتقال إلى نهاية النافذة الحالية
PgUp	الانتقال إلى الأعلى نافذة واحدة
PgDn	الانتقال إلى الأسفل نافذة واحدة
Ctrl-PgUp	الانتقال إلى بداية المذكرة
Ctrl-PgDn	الانتقال إلى نهاية المذكرة
Enter	الانتقال إلى بداية السطر التالي
Delete	حذف الحرف الواقع عند المؤشر الحالي
BkSp	حذف الحرف الذي على يسار المؤشر
Tab	إقحام (حشر) حرف جدولة (مسافات فارغة)
أي حرف قابل للطباعة	إقحام (حشر) حرف في موقع المؤشر الحالي
Ctrl-Y	حذف السطر الحالي بالكامل
Ctrl-T	حذف كلمة من اليمين
Ctrl-B	إعادة تنسيق الفقرة الحالية
Ctrl-V or Insert	التبديل بين الإقحام والكتابة فوقية
Ctrl-W	الخروج من ( ) MEMOEDIT وحفظ التعديلات
Esc	الخروج من ( ) MEMOEDIT وتجاهل التعديلات



يبين المثال التالي كيفية استخدام هذه المفاتيح:

```
memoedit(notes, 10, 10, 12, 69, .f.) // display, allow user to scroll
memoedit(notes, 10, 10, 12, 69, .f., .f.) // display & continue
notes := memoedit(notes, 10, 10, 12, 69) // allow user to edit
```

## وظيفة قراءة المذكرة ( MEMOREAD )

تتمكنك هذه الوظيفة من قراءة محتويات ملف داخل سلسلة حرفية ، إذ تقبل أسم الملف كمتغير. ويجب ألا يزيد طول الملف عن ٦٥٥٣٥ حرفاً. وتجدر الإشارة إلى أنه إذا لم تتمكن هذه الوظيفة من العثور على الملف المحدد في الدليل الحالي فإنها ستبحث في مسار DOS ، وإذا لم تستطع العثور عليه في أي مكان على الإطلاق فستعرض أن قيمة الملف هي سلسلة صفرية (أي أنها لم تجد الملف فعلاً ، أو أن الملف فارغ تماماً).

### اعتبارات خاصة للشبكات

إن الوظيفة ( MEMOREAD ) تحاول فتح الملفات على أنها مشتركة وللقراءة فقط ، فإذا كان الملف مفتوحاً من قبل أحد الأشخاص المحددين للتعامل معها فسترجع هذه الوظيفة سلسلة قيمتها صفر ، أي خالية.

## الوظيفة ( MEMOWRITE )

هذه الوظيفة هي بعكس سابقتها ( MEMOREAD ). فهي تكتب سلسلة من الحروف (أو مذكرة) في ملف النص ، والتركيب اللغوي لها هو:

MEMOWRITE ( <cFile> , <cString> )

حيث أن المتغير <cFile> اسم الملف المكتوب. وإذا لم تحدد المسار فستتم كتابة الملف في الدليل الحالي في دوس. وأما إذا كان الملف موجوداً فستتم الكتابة فوقه. ويمكن

استخدام الوظيفة File( ) للتأكد من وجود الملف قبل الكتابة. وأما المتغير <cString> فهو سلسلة حرفية أو المذكرة المطلوب استخدامها لدى إنشاء الملف. وسُرجع هذا المتغير القيمة المنطقية "حقيقي" T. إذا تمت الكتابة بشكل ناجح ، و "غير حقيقي" F. إذا لم تنجح الكتابة.

ويبين السطر التالي كيفية كتابة الوظيفة MEMOWRITE( ) و MEMOREAD( ) و MEMEDIT( ) لتحرير ملف نص:

```
memowrit( "temp.txt", memoedit(memoread( "temp.txt" ),  
0, 0, maxrow( ), maxcol( ) ) ) ;
```

## الوظيفة MLCOUNT( )

تعطيك هذه الوظيفة العدد الإجمالي للأسطر في السلسلة الحرفية أو المذكرة. ويمكن استخدام هذه المعلومات من قبل الوظيفة MEMLINE( ) لإنشاء حلقة وعرض المذكرة. والتركيب اللغوي لاستخدام هذه الوظيفة هو:

```
MLCOUNT( <cString>, [<nLength>], [<nTab>], [<IWrap>] )
```

حيث أن المتغير <cString> هو السلسلة المقصودة ، والمتغير <cLength> عدد حروف كل سطر وهو يتفاوت ما بين ٤ - ٢٥٤ حرفاً ، والرقم المفترض هو ٧٩.

أما المتغير <nTab> فهو عدد المسافات المحدد لكل مسافة جدولة أمامية. والقيمة المفترضة هي ٤ مسافات ، وهي قيمة ثابتة تقريباً.

أما المتغير <IWrap> فيشير إلى أنك ستستخدم طريقة لف السطور أم لا. فإذا مررت القيمة المنطقية "غير حقيقي" F. فلن يتم استخدامه. ويفترض تشغيل طريقة لف السطور تلقائياً.

وترجع الوظيفة ( ) MLCOUNT عدد السطور الإجمالي إلى الطول المحدد في السلسلة أو المذكرة ، ويرجى مراجعة الأمثلة المتعلقة باستخدامات هذه الوظيفة.

## الوظيفة ( ) MEMOLINE

تستخرج هذه الوظيفة سطراً واحداً من النص من السلسلة الحرفية أو المذكرة وغالباً ماتستخدم بالتزامن مع الوظيفة ( ) MLCOUNT لإخراج مذكرة كاملة. والتركيب اللغوي لهذه الوظيفة مماثل جداً للوظيفة المذكورة.

MEMOLINE( <cString>, [<nLength>], [<nLine>], [<nTab>], ;  
[<IWrap>] )

فالمتغير <cString> هو السلسلة أو المذكرة المراد استخراجها.

أما المتغير <nLength> فهو يمثل عدد الحروف لكل سطر ، ويمكن أن يبلغ نطاقه ما بين ٤ إلى ٢٥٤ حرفاً. والعدد الافتراضي للنظام هو ٧٩.

أما المتغير <nLine> فهو يحدد السطر المراد استخراج السلسلة منه. والسطر الافتراضي هو رقم واحد.

أما المتغير <nTab> فهو يحدد عدد المساحات المستخدمة للجدولة. القيمة الافتراضية ٤.

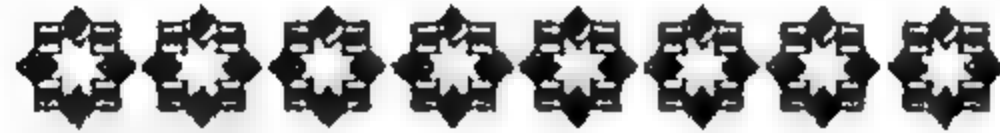
أما المتغير <IWrap> وهي تشير ما إذا كان التفاف الكلمات مستخدماً أم لا. فإذا مررت القيمة المنطقية "خاطئء" (F.) فإن هذه الوظيفة لن تستخدم. أما في الوضع الافتراضي فإن التفاف الكلمات يعتبر نشطاً.

ترجع الوظيفة ( ) MEMOLINE سطر النص المحدد من السلسلة أو المذكرة. وأما إذا كان السطر يحتوي على عدد أقل من الحروف المحددة لطوله فسوف تضاف إليه

مسافات فارغة بشكل مناسب في نهايته. وأما إذا كان عدد حروفه أكبر من السطور المتوفرة في السلسلة أو المذكورة ، فإن الوظيفة سترجع سلسلة خالية.

ويبين الجزء التالي من البرنامج كيفية طباعة أو عرض مذكرة كاملة باستخدام هذه الوظيفة:

```
nLength := 60
nLines := mlcount(customer->text, nLength)
for x := 1 to nLines
  ? memoline(customer->text, nLength, x)
next
```



## المخرجات Output

بعد أن تم إدخال كمية كبيرة من المعلومات من خلال شاشات إدخال البيانات الرائعة، يحتمل أنك تريد إخراجها إلى حيز الوجود. يوفر كليب 5.2 العديد من الوسائل والأدوات المختلفة لإخراج البيانات. فيمكنك إرسال هذه البيانات إلى الطابعة أو ملف نص أو ملف قاعدة بيانات.

## الطابعة

### الوظيفة SET DEVICE

يمكنك هذا الأمر من الانتقال ما بين الشاشة والطابعة. ويفترض أن تكون الشاشة هي الوسيلة النشطة. والتركيب اللغوي لهذا الأمر سهل جداً:

SET DEVICE TO SCREEN

أو

SET DEVICE TO PRINTER

### الوظيفة SET PRINTER( )

عند تشغيل هذه الوظيفة يتوقف عمل الأمرين ؟ و ؟؟ بحيث ترسل المخرجات إلى الطابعة. انظر فقرة SET CONSOLE لمشاهدة أمثلة عن استخداماتها. ويمكن استخدام هذه الوظيفة بشكل اختياري أيضاً لإعادة توجيه إخراج الطابعة إلى ملف نص. ويمكن التعبير عن اسم الملف حرفياً أو باستخدام متغير ما يوضع بين قوسين كما هو مبين أدناه:

```
cFile := "output.txt"
set printer to temp.txt
set printer to (cFile)
```

## ملاحظة

إذا أعدت توجيه مخرجات الطابعة إلى ملف نص ، ثم استخدمت الأمر SET DEVICE TO PRINT فإن كل SAY..@ ستوجه إلى ذلك الملف أيضاً.

إذا استخدمت هذا الأمر لإعادة التوجيه ، فقد ترغب أيضاً باستخدام الفقرة ADDITIVE. حيث توجه هذه الفقرة برنامج كليبر لإلحاق النص المطلوب في الملف المطلوب. أما إذا لم تستخدم هذه الفقرة ، فإنه ستم الكتابة على الملف السابق .overwrite

## الأمر SET CONSOLE

يقوم هذا الأمر بتشغيل كافة الأوامر المتعلقة بالشاشة. فإذا أردت استخدام كل من أمري ؟ و ?? للطباعة ، ولكن دون أن تشاهد المخرجات على الشاشة ، يمكنك استخدام التركيب اللغوي التالي:

```
set cinsole off
set printer on
do while .not. eof( )
    ? field1, field2, filed3
    dbskip( )
enddo
set printer off
set cinsole on
```

وسيقوم الأمر SET DEVICE TO PRINT بتوجيه أي مخرجات باستخدام أمر SAY..@ إلى الطابعة. وتجدر الإشارة إلى أنه يتم تجاهل الفقرة الاختيارية للون COLOR في مثل هذه الحالات.



## الأمر ؟ و ??

إذا تم ضبط تشغيل الطابعة SET PRINT ON على وضعية التشغيل فإن مخرجات أي من أمري ؟ و ?? سيتم توجيههما إلى الطابعة. ولاحظ أن عبارة اللون الاختيارية سيتم تجاهلها أيضاً كسابقتها في هذه الحالة.

## إخراج الورق EJECT

يرسل هذا الأمر رمز يطلب من الطابعة تغذيتها بورقة (وهذا الرمز هو الآسكي ١٢). كما يقوم هذا الأمر بإعادة تجهيز رأس الطابعة والسطر والعمود إلى الموقع صفر. وقد ترغب هنا بإصدار هذا الأمر عندما تصل إلى نهاية كل صفحة (والذي يمكن التعرف عليه من خلال اختبار الوظيفة ( ) PROW).

وبالإضافة إلى ذلك ، فيجب الانتباه إلى أنك إذا عنونت سطر الطابعة على أنه أقل من موقع السطر الأخير فإن كليبر سيلفظ الصفحة مباشرة. لذلك يجب الطابعة من اليسار إلى اليمين ومن الأعلى إلى الأسفل. وبين جزء البرنامج التالي هذه الطريقة:

```
set device to print
@ 2, 1 say date( )
@ 1, 1 say time( )      // forces a page eject
```

## وظيفة ( ) PCOL و ( ) PROW

تشبه هاتان الوظيفتان كلاً من وظيفتي ( ) ROW و ( ) COL ماعدا أنهما يرجعان إلى رأس الطابعة بدلاً من المؤشر. وتعتمد هذه الوظيفة على الإحداثيات التي تبدأ من الصفر. وكما هو الحال في إحداثيات الشاشة في كليبر ، فإن هاتين الوظيفتين تعتمدان على الصفر (فالسطر الأول هو صفر وليس واحداً).

ويتم تحديث هاتين الوظيفتين إذا تم ضبط الجهاز DEVICE على الطابعة فقط (set device to printer أو set printer on). وعند البدء بتشغيل أي برنامج فسيكون ضبطهما على الصفر.

وتعيد الوظيفة EJECT ضبط كل من ( ) PCOL و ( ) PROW إلى الصفر. وتضبط عمليات الطابعة ( ) PROW إلى آخر سطر تمت الطابعة عليه ، بينما تعمل وظيفة ( ) PCOL إلى ضبط عمود واحد بعد آخر عمود تمت الطابعة عليه. ويمكن إعادة ضبط هاتين الوظيفتين باستخدام الوظيفة ( ) SETPRC.

#### تنبيه

إذا كنت تستخدم حرفي آسكي الخاصين بالتغذية السطرية line-feed وتغذية الصفحة form-feed (وهما 10 و 12 على التوالي) ، فإن قيمة الوظيفة ( ) PROW لن يتم تحديثها.

#### تحذير

إذا أرسلت ما يسمى بـ : متتالية الهروب Escape Sequences إلى الطابعة سينفذ ( ) PCOL تناغمه لأن كليبر يعد الرموز التي أرسلت إلى الطابعة وليس لديه أية وسيلة ليعرف أنها طبعت أم لا . ويمكن تجنب حدوث مثل هذه المشاكل باستخدام ( ) SETPRC كما سنبين أدناه.

```
function printit
set device to print
use customer
do while ! eof( )
    @ prow( )+1, 1    say customer->first
```

```
@ prow( ), pcol( )    say customer->last
@ prow( )+1, 1        say customer->address
@ prow( )+1, 1        say rtrim (customer->city)
@ prow( ), pcol( )+1  say customer->state
@ prow( ), pcol( )+1  say customer->zip
if prow( ) > 57
    eject
endif
enddo
set device to screen
return nil
```

## وظيفة SETPRC( )

تعيد هذه الوظيفة تجهيز القيم الداخلية لكل من PROW( ) و PCOL( ). وتعيد دوماً القيمة ، صفر كما ذكرنا آنفاً. وتفيد هذه الوظيفة جداً في إرسال أوامر تحكم إلى الطابعة وتعيد تجهيز موقع رأس الطابعة. كما يمكن استخدام الوظيفة SETPRC( ) لتجميد عملية إخراج الورق eject عندما تكون الطابعة مستمرة مع الأمر SAY..@ (ويدويا يتم ذلك بإعادة تجهيز قيمة الوظيفة PROW( )). ويوضح المثال التالي بشكل جيد كيفية استخدام الوظيفة SETPRC( ) لمتاليات الهروب escape sequences:

```
set device to print
@ 1, 5 say "normal"
@ prow( ), pcol( ) + 2 say printcode(chr(15)) + "compressed"
@ prow( ), pcol( ) + 2 say printcode(chr(18)) + "normal again"
set device to screen
```

```
function printcode(cString)
local oldrow := prow( )
local oldcol := pcol( )
local oldprint := set (_SETPRINTER, .T.)
qqout (cString)
setprc (oldrow, oldcol)
set (_SET_PRINTER, oldprint)
return nil
```

## ملف آسكي ASCII

هناك ثلاث طرق لطباعة المخرجات على ملف آسكي وهي: (١) باستخدام أمر SET PRINT TO لإعادة توجيه إخراج الطابعة . (٢) باستخدام أمر COPY TO. (٣) باستخدام أمر SET ALTERNATE.

## الأمر COPY TO

يُنشئ هذا الأمر إما ملف قاعدة بيانات (DBF) أو ملف نصوص ، وطريقة استخدام هذا الأمر على النحو التالي:

```
COPY [FIELDS <field list>] TO <file>
      [<SCOPE>] [WHILE <condition>] [FOR <condition>]
      [SDF | DELIMITED [WITH BLANK | <delimiter>] ]
```

حيث تمثل <File> الملف الهدف. إذا عيّنت الفقرة الاختيارية SDF أو DELIMITED ، فإنه يفترض أن يكون ملحق الملف TXT. ، وخلاف ذلك فإن DBF هو الملحق الافتراضي للملف. أما الفقرة الاختيارية FIELDS يمكن استخدامها للاقتصار على الحقول المستخدمة لإنشاء ملف قاعدة بيانات جديدة DBF..

أما المتغير <Scope> فهو يعرف مجال ملف قاعدة البيانات المراد نسخه. ويفترض المطلوب نسخ جميع السجلات. والطريقة الشائعة لاستخدام هذه الفقرة هي: NEXT <n> ، حيث أن <n> تمثل عدد السجلات المراد نسخها ابتداءً من السجل الحالي.

أما الفقرتان FOR و WHILE فهما تسمحان لنا بتوصيف التعبيرات المنطقية لتحديد نطاق السجلات المراد إخراجها. وتبين سطور البرنامج التالية طريقتين لإخراج كل السجلات التي تحمل الاسم "Sulaiman Al-maiman". فإذا كنت تستطيع استخدام الفهرسة للقفز إلى أول سجل ممكن ، فإن الفقرة WHILE ستكون عامل سرعة هامة في برنامجك.

```
use customer
copy to temp.txt sdf for customer->name == "JOHN SMITH"
// using index
use customer index customer
if dbseek( "JOHN SMITH" )
    copy to temp.txt sdf while customer-name == "JOHN SMITH"
endif
```

كما تقوم الفقرتان الاختياريتان SDF و DELIMITED بتصميم نوع مخرجات آسكي وتعتبر هاتان الفقرتان مفيدتين إذا كنت تخطط لنقل البيانات إلى بيئة تشغيل أخرى.

كما أن مخرجات الملفات SDF هي سجلات ذات طول ثابت ، ويفصل كل منها عن الآخر برمز الرجوع carriage return أو رمز تغذية سطر line feed (وهما الآسكي ١٠ و ١٣ على التوالي). ويتم إضافة مسافات مناسبة لملء الفراغات ، بينما يتم ملء الحقول الرقمية بمسافات فارغة قبلها. وأما حقول التاريخ فتكتب بالشكل "yyyymmdd" وأما الحقول المنطقية فتكتب إما "T." أو "F." وأما رمز نهاية الملف في آسكي فهو ٢٦.

أما باستخدام DELIMITED فإنه قد يتغير طول السجلات ، ويفصل بين سجل وآخر برمز الرجوع carriage return أو رمز تغذية سطر line feed. في حين أن الحقول مختلفة الطول وتفصل بالفاصلة. وتحجز الحقول الحرفية بفاصلتين. والفواصل المفترضة هي علامات التنصيص " " والتي يمكن تغييرها باستخدام الفقرة: WITH <delimiter> ، وأما المسافات الفارغة والتي تسبق كلاً من الحقول الرقمية والحرفية فستحذف. وكما هو الحال بالنسبة لمخرجات SDF ، فإنه يتم كتابة حقول التاريخ بالشكل "yyyymmdd" ، أما الحقول المنطقية فتكتب إما "T" أو "F". أما رمز نهاية الملف فهو آسكي ٢٦.



## أمر "جهاز بديلاً" SET ALTERNATE

يشغل هذا الأمر ، ويوقف عن التشغيل ما إذا كان إخراج كل من أمري ؟ و ?? يجب إصدارهما إلى ملف نص إضافي. فهو يسمح بالطباعة والنسخ إلى ملف نص في الوقت ذاته. ويستخدم التركيب اللغوي ذاته المستخدم لتجهيز الطابعة SET PRINTER TO كما تشاهد أدناه:

```
cFile := "output.txt"
set alternate to temp.txt
set alternate to (cFile)
```

وكما هو الحال مع الأمر SET PRINTER ، فإنه يمكنك استخدام الفقرة ADDITIVE والتي تنسب في أن يقوم كليبر بإلحاق المخرجات إلى الملف إذا كان موجوداً. أما إذا لم تستخدم الفقرة ADDITIVE ، فإنه سيتم الكتابة على ملف الهدف.

## ملف قاعدة البيانات DBF

إن أسهل طريقة لنسخ ملف قاعدة بيانات DBF إلى آخر هو استخدام أمر انسخ ملف COPY FILE كما هو الحال في DOS. إلا أنك إذا أردت مزيداً من التحكم بهذه العملية فاستخدم أمر COPY TO ، كما هو موضح أعلاه دون استخدام الخيارين SDF أو DELIMITED. كما يمكنك استخدام الفقرة الخيائية FIELDS لتحديد المخرجات إلى حقول معينة (الوضع الافتراضي ، أن كل الحقول الموجود ملف قاعدة البيانات المصدر ستستخدم).

ويمكن إنشاء ملف قاعدة بيانات جديد مثلاً باسم CUST.DBF يحتوي فقط على الحقلين الأول والأخير والسجلات الموجودة في مجال محدد مثلاً ، كالمنطقة الشرقية ، مثال:

```
use customer
copy fields first, last to cust for customer->state == "OR"
```



أما إذا أردت إنشاء ملف قاعدة بيانات يحتوي على بيانات تم حسابها ، فيمكن استخدام الوظيفة ( ) DBCREATE (وهي مشروحة أدناه) ، ثم إضافة سجلات جديدة لها إذا لزم الأمر.





## متفرقات

### عمليات الملف

يقدم كليب بعض الوظائف والأوامر التي تنفذ بعض عمليات ملفات أوامر دوس.

#### أمر "التجهيز الافتراضي" SET DEFAULT

يفترض كليب بهذا أن كافة الملفات اللازمة موجودة في الدليل الحالي ، وعلاوة على ذلك فإن كافة الملفات التي يتم إخراجها من قبل البرنامج المستخدم سيتم كتابتها أيضاً في الدليل الحالي.

إلا أنه يمكن تغيير هذه الطريقة باستخدام هذا الأمر SET DEFAULT. إذ يوجه هذا الأمر برنامج كليب أن يبحث عن الملفات ويكتبها في دليل محدد. فعلى سبيل المثال SET DEFAULT TO c:\data فهذا يغير الدليل الافتراضي ليصبح c:\data. وإن استخدام هذا الأمر دون إضافة أية معايير أخرى له سيعيد تجهيز الدليل المفترض بحيث يصبح الدليل هو الدليل الحالي لـ: DOS.

#### ملاحظة

وتجدر الإشارة إلى أن هذا الأمر لا يغير الدليل الحالي في دوس ، كما أن وظائف ملفات كليب الدنيا low-level file functions (والتي يتعدى موضوعها نطاق بحثنا هنا) لا تتأثر بهذا الأمر أيضاً.

## الوظيفة File( )

تستخدم هذه الوظيفة لتحديد ما إذا كان الملف يوجد فعلاً أو لا يوجد. وتقرر إليه اسم الملف المطلوب (مع المسار إذا لزم الأمر) ، وسرّج قيمة منطقية "حقيقي" أو "غير حقيقي" إذا كان الملف موجوداً فعلاً أو لا.

## الوظيفة COPY FILE

هي وظيفة لنسخ ملف إلى آخر ، وتركيبها اللغوي هو كما يلي:

COPY FILE <source> TO <target>

ويمكن تحديد كل من الملف المصدر "source" و الهدف "target" بقيم حرفية ، أو يمكن استخدام متغيرات إذا كانت تتضمن أسماء ملفات ، وتُحاط ضمن قوسين ، على النحو التالي:

```
cfile1 := "somfile.txt"
cfile2 := "newfile.txt"
copy file source.dbf to target.dbf
copy file (cfile1) to (cfile2)
```

### ملاحظة

إذا كان ملف الهدف <target> موجوداً أصلاً ، فإنه ستم الكتابة أعلاه. ومن ناحية أخرى ، فقد ترغب في استخدام الوظيفة FILE( ) للتأكد من عدم وجود الملف قبل المضي في إصدار الأمر COPY FILE.

## الوظيفة ( ) FEREASE

تسمح هذه الوظيفة ملفاً محدداً يصدر الأمر إليها بمسحها. وسوف ترجع القيمة "صفر" إذا تمت عملية المسح بنجاح ، أو القيمة "-1" إذا وقع خطأ في العملية. وفي حالة وقوع الخطأ فإن الوظيفة ( ) FEERROR يمكنها أن تحدد الخطأ بالضبط.

المثال التالي يوضح كيفية استخدام الوظيفة السابقة:

```
if ferase( "temp.txt" ) == -1
    alert( "could not erase file. . . error " + ;
        ltrim(str(ferror( ) ) ) )
endif
```

### تحذير

يجب التأكد من إغلاق الملف قبل مسحها.

## الوظيفة ( ) FRENAME

تتيح هذه الوظيفة إمكانية إعادة تسمية ملف باسم جديد. فإذا كان الملف موجوداً فستحقق هذه العملية وترجع قيمة "صفر" كسابقاتها ، فسترجع القيمة "صفر" إذا تمت عملية إعادة التسمية بنجاح ، أو "-1" إذا حدث خطأ. وفي حالة حدوث خطأ ، فإن الوظيفة ( ) FERROR يمكنها الانسحاب وطلب تحديد اسم آخر.

والمثال التالي يوضح كيفية استخدام الوظيفة ( ) FRENAME:

```
if rename( "temp.dbf", "customer.dbf" ) == -1
    alert( "could not rename file. . . error " + ;
        ltrim(str(ferror( ) ) ) )
endif
```

## الوظيفة ( ) FERROR

في حال وجود خطأ يتعلق بالملف ، مثل المسح ( ) FERASE و إعادة التسمية ( ) FRENAME ، فيمكن أن تعطيك هذه الوظيفة مزيداً من المعلومات عن سبب هذا الخطأ الذي حدث.

وإن القيم الراجعة المحتملة نتيجة استخدام الوظيفة ( ) FERROR هي كما يلي:

الرقم	التوضيح
0	نجاح
2	الملف غير موجود
3	الممر غير موجود
4	عدد كبير من الملفات مفتوح
5	يرفض الوصول
6	معالجة غير صحيحة
8	الذاكرة غير كافية
15	المشغل المحدد غير صحيح
19	الوسيط محمي من الكتابة
21	المشغل غير جاهز
23	بيانات CRC خاطئة
29	غير قادر على الكتابة
30	غير قادر على القراءة
32	المشاركة بالقوة
33	اقفال المشاركة



## إنشاء ملف قاعدة بيانات بشكل سريع On-The-Fly

تقوم الوظيفة ( ) DBCREATE بكل سهولة بإنشاء ملفات قاعدة البيانات برمجياً. حيث يتم إنشاء هيكل ملف قاعدة بيانات من مصفوفة متداخلة المصفوفات تحتوي على معلومات عن الحقول المطلوبة. وفي المثال التالي ، قمنا بإنشاء ملف قاعدة بيانات اسمه ITEM.DBF والمكون من حقلين اثنين وهما: ITEM (وطوله ٢٠ حرفاً) و DESCRIP (وطوله ٤٥ حرفاً).

```
dbcreate( "items", { ;
                { "item", "C", 20, 0 }, ;
                { "descrip", "C", 45, 0 } ;
            } )
```

### تنبيه

حتى وإن لم يكن الحقل رقمياً non-numeric ، إلا أنه يجب التأكد من كتابة الصفر في الموقع الرابع لكل مصفوفة متداخلة. خلاف ذلك ، فإن الوظيفة ( ) DBCREATE ستنشئ الملف DBF. ممثلاً بالأخطاء في ترويسة الملف.

## تعديل هيكل ملف قاعدة البيانات بشكل سريع On-The-Fly

تعمل الوظيفة ( ) DBSTRUCT بعكس الوظيفة ( ) DBCREATE تماماً. فهي ترجع مصفوفة تحتوي على مصفوفات متداخلة تتضمن معلومات عن البنية الهيكلية للملف قاعدة البيانات الحالي. وستحتوي المصفوفة على عنصر واحد لكل حقل في قاعدة البيانات ، وكل عنصر من تلك العناصر هو مصفوفة تتألف من أربعة عناصر تطابق الحقول على الترتيب التالي:

١ - Name

٢ - Type

Length -٣

Decimals -٤

ويبين المثال التالي كيفية تعديل هيكل ملف قاعدة بيانات بسرعة بطريقة البرمجة وباستخدام وظيفة ( DBCREATE ) و ( DBSTRUCT ) ، فهي تفتح أية قاعدة بيانات وتضيف الحقل DUMMY إليها:

```
function test(dbf_file)
local a
use (dbf_file)
a := dbstruct( )
aadd(a, { "D ", "C", 10, 0 } )
dbcreate("new", a)
use new
append from (dbf_file)
use
/ /----- look for existing backup files, and delete them if exist
if file(dbf_file + '.tbk' )
    ferase(dbf_file + '.tbk' )
endif
fename(dbf_file + ".dbf", dbf_file + ".bak")
if file(dbf_file + ".dbf")
    frename(dbf_file + ".dbf", dbf_file + ".tbk")
endif
frename('new.dbf' dbf_file + ".dbf")
if file("new.dbf")
    frename('new.dbf' dbf_file + ".dbf")
endif
return nil
```

## استرجاع معلومات دليل

إن الوظيفة ( DIRECTORY ) في كليبر هي ممتازة لاسترجاع معلومات عن الدليل المطلوب. وترجع هذه الوظيفة مصفوفة ذات مصفوفات متداخلة تحتوي على معلومات

عن كل ملف مطابقة لمواصفات الملف. وسيكون لكل من المصفوفات المتداخلة الهيكل العام التالي:

١- اسم الملف (filename)

٢- الحجم (size)

٣- التاريخ (date)

٤- الوقت (time)

٥- خواص (attributes)

وإليك مثلاً على هذه الوظيفة ، حيث تقوم بعرض كل ملفات المكتبة الخاصة بكليبر والموجودة في الدليل LIB:

```
#include "directry.ch"
```

```
function test
```

```
local f_ := directory("c:\clipper\lib\*.lib")
```

```
local num_files := len(f_)
```

```
local x
```

```
for x := 1 to num_files
```

```
    qout( padr(f_[x][F_NAME], 14), f_[x][F_SIZE], ;
```

```
        f_[x][F_DATE], f_[x][F_TIME], f_[x][F_ATTR] )
```

```
next
```

```
return nil
```

وسيكون الإخراج مشابهاً لما يلي:

CLIPPER	LIB	510,097	02-15-93	5:20a
CLI	LIB	80,719	02-15-93	5:20a
DBFNTX	LIB	38,465	02-15-93	5:20a
DBFNDX	LIB	27,175	02-15-93	5:20a
DBFCDX	LIB	103,843	02-15-93	5:20a
DBFMDX	LIB	75,861	02-15-93	5:20a
DBPX	LIB	170,645	02-15-93	5:20a
EXTEND	LIB	125,881	02-15-93	5:20a
RTLUTILS	LIB	53,925	02-15-93	5:20a
TERMINAL	LIB	14,369	02-15-93	5:20a
ANSI TERM	LIB	12,321	02-15-93	5:20a
ROUTER	LIB	13,857	02-15-93	5:20a
PCBIOS	LIB	14,369	02-15-93	5:20a
SAMPLE*	LIB	53,891	02-15-93	5:20a

## اختيار الملفات باستخدام DIRECTORY() و ACHOICE()

يبين المثال التالي كيفية فتح مربع فقاعي pop up (ينفتح مباشرة) على الشاشة باستخدام الوظيفة ACHOICE() ، ليختار منها المستخدم أي نوع من الملفات يريد. فإذا لم يتم تمرير مواصفات الملف إلى هذه الوظيفة فسيفترض أنك تريد فتح ملف قاعدة بيانات ..DBF

```
#include "box.ch"
#include "directry.ch"

function pickfile(cFilespec)
local oldscm := savescreen(5, 33, 19, 46)
local oldcolor := setcolor( '+w/b' )
local oldrow := row( )
local oldcol := col( )
local f_
local num_files
local x
if cFilespec == NIL
    cFilespec := "+.DBF"
endif
f_ := directory(cFilespec)
num_files := len(f_)
for x := 1 to num_files
    f_[x] := f_[x][F_NAME]
next
@ 5, 33, 19, 46, box B_SINGLE + ''
x := achoice(6, 34, 18, 45, f_)
restscreen(5, 33, 19, 46, oldscm)
setcolor(oldcolor)
setpos(oldrow, oldcol)
return if (x > 0, f_[x], NIL)
```

## نظام التاريخ والتوقيت

يمكن استرجاع كل من التاريخ والتوقيت الموجودين في الجهاز باستخدام أمري DATE() و TIME(). وترجع وظيفة التاريخ بيانات التاريخ على صيغة بيانات تاريخ

والتي يمكن فيما بعد عرضها باستخدام وظيفة التحويل ( ) DTOC. حيث تقوم الوظيفة ( ) DTOC بتحويل أي تعبير تاريخ إلى سلسلة حرفية بالشكل التالي: "MM/DD/YY" ويمكن تغيير طريقة العرض حسب الطلب.

وترجع وظيفة التوقيت ( ) TIME الوقت في سلسلة حرفية على الشكل التالي: "hh:mm:ss". وهذا النوع من التوقيت يدعى التوقيت العسكري (وهو المكون من ٢٤ ساعة)، وعلى الرغم من ذلك يمكنك وبكل سهولة تحويله إلى التوقيت بنظام ١٢ ساعة كما هو الحال في البرنامج التالي:

```
function ampm( cTime )
if val(cTime) < 12
    cTime += " am"
elseif val(cTime) == 12
    cTime += " pm"
else
    cTime := str(val(cTime) - 12, 2) + substr(cTime, 3) + " pm"
endif
return cTime
```

أما إذا أردت حساب وقت معين فيفضل استخدام الوظيفة ( ) SECONDS إذ تبين الثواني التي انقضت منذ منتصف الليل.

## استرجاع البيئة

يمكن استرجاع محتويات أي متغير بيئة في DOS باستخدام الوظيفة ( ) GETENV التي تقبل مصفوفة حساسة للسلسلة الحرفية التي تمثل اسم متغير البيئة المراد استرجاعه. فإذا كان هذا المتغير موجود، فإن الوظيفة ترجع قيمته، وخلاف ذلك، فإنها سترجع سلسلة صفيرة.

وفيما يلي نبين كيفية استخدام هذه الوظيفة على شبكة عمل محلية تحتوي على دليل مخصص لكل مستخدم، ويفترض أنك حددت اسم مستخدم لكل من المستخدمين.

```
set default to ( "F:\USERS\" + getenv( "userid" ) )
```



## ملخص

يمكنك الآن البدء بكتابة برنامجك باستخدام برنامج كليبر 5.2 والتي يمكنها إدارة قواعد البيانات. وكذلك ستشعر باطمئنان وثقة تامة لدى التعامل مع مختلف الوظائف المتعلقة بقواعد البيانات. ويجب الإحاطة بشكل تام بالعوامل المستخدمة في كليبر ، وكذلك كتل التحكم. وكذلك يجب أن تلم بالمبادئ الأساسية لكيفية إقفال الملفات والسجلات بحيث تستخدم هذه في برامج يستخدمها عدد من المستخدمين.

وأخيراً ، يجب أن تتذكر جميع برنامجك دوماً باستخدام خيار V/V واستخدام دائماً الإعلانات المحلية أو الساكنة Local / Static للمتغيرات التي تستخدمها. وتذكر أن تسبق حقول قاعدة البيانات بالعوامل المناسبة لها وإستخدم وظائف db\* بدلاً من استخدام عبارة SELECT. لاتنس بتاتاً أن تبرمج باستخدام طريقة "وحدات البرمجة".









● الربط باستخدام

RTLINK

وكذلك BLINKER

● جزء كامل يوضح

أساسيات البرمجة

● باستخدام كليبر 5.2

● جزء كامل للحديث

عن البرمجة المتقدمة

● باستخدام Tbrowse

وكذلك TBColumn

● فصل كامل عن كتل

الشفيرة

Code Blocks

● مئات الأمثلة

وعشرات الجداول

التوضيحية والفوائد

والأفكار والتحذيرات

● بالإضافة إلى العديد

من المميزات المتناثرة

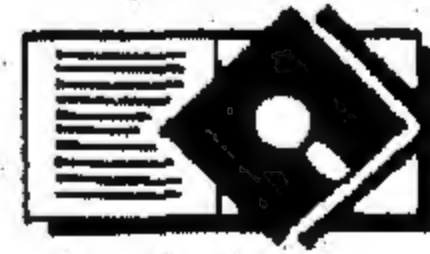
في مواضع مختلفة

من الكتاب.

## مختصرات كليبر

- الكتاب الوحيد في العالم العربي الذي يشرح كليبر 5.2 ، بالإضافة إلى الكم الهائل من المعلومات التي يشرحها الكتاب من خلال أجزائه الثلاث.
- تم تقسيم الكتاب إلى ثلاث أجزاء هي: مقدمة البرمجة ، أساسيات البرمجة ، البرمجة المتقدمة.
- شرح لمعظم أوامر وتعليمات ووظائف كليبر الأساسية للإصدار 5.2.
- جزء خاص عن مقدمة البرمجة بلغة كليبر ماهي؟ وكيف يمكنك الاستفادة منها؟
- تصميم وإنشاء وكتابة أقوى التطبيقات الاحترافية باستخدام لغة كليبر 5.2.
- فصل موسع لطريقة استخدام برنامج كشف الأخطاء Debugger بأسلوب سهل ومبسط.
- كما خصص مقدار كبير من الجزء الثالث للحديث عن البرمجة باستخدام Tbrowse و TBColumn ومزاياها المفيدة في استعراض قواعد البيانات.
- كما تم شرح كتلة الشيفرة بأسلوب سهل ، يجعل من هذه التقنية الجديدة في كليبر مريحة وسهلة الاستخدام.
- كما خصص فصل للحديث عن استراتيجيات عمل الشبكة نوفيل مع كليبر 5.2
- فصل كامل للحديث عن مفاتيح التجميع والربط.
- شرح للمعالج الأولي Preprocessor وملفات الرويسة والموجهات وغير ذلك.
- فصل كامل لشرح طريقة إعلان المتغيرات بجميع أنواعها بأسلوب سهل وميسر.
- فصل كامل لشرح طريقة استخدام المصفوفات وكذلك الوظائف المتعلقة بها.
- شرح طريقة التحويل إلى نظام التشغيل MS-DOS باستخدام الرابط BLINKER 2.0
- شرح لطريقة تحويل برامجك من شيفرة المصدر source code إلى برامج قابلة للتنفيذ EXE. تعمل بشكل مستقل.
- فصل كامل يوضح طريقة تصميم واجهة المستخدم والأدوات التي يوفرها كليبر للقيام بهذه المهمة في توفير شكل جمالي ومريح للمستخدم.

كتاب



قرص

Bibliotheca Alexandrina



0339890